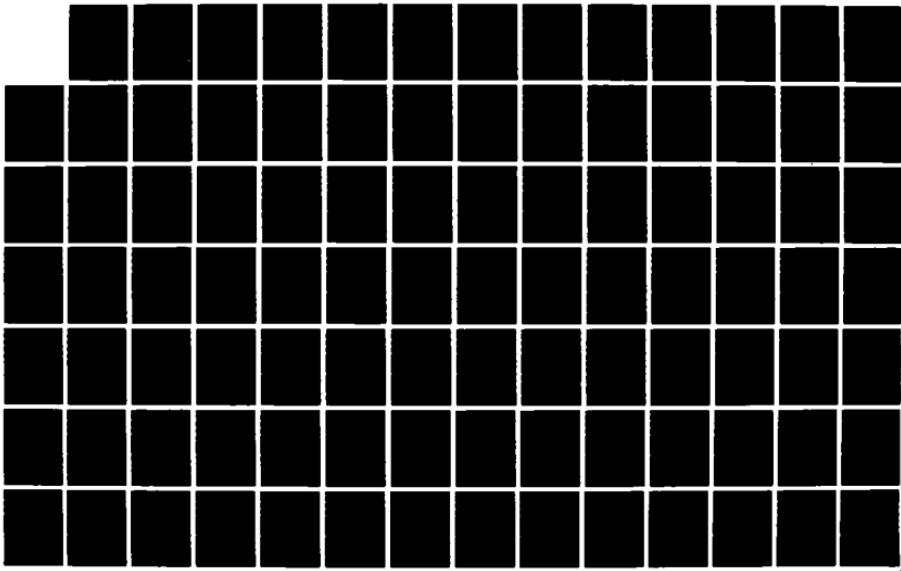
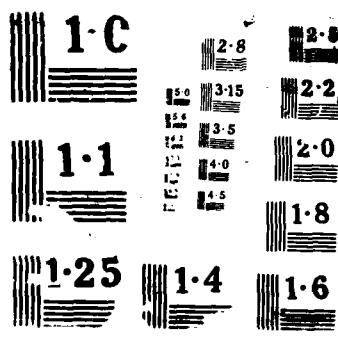


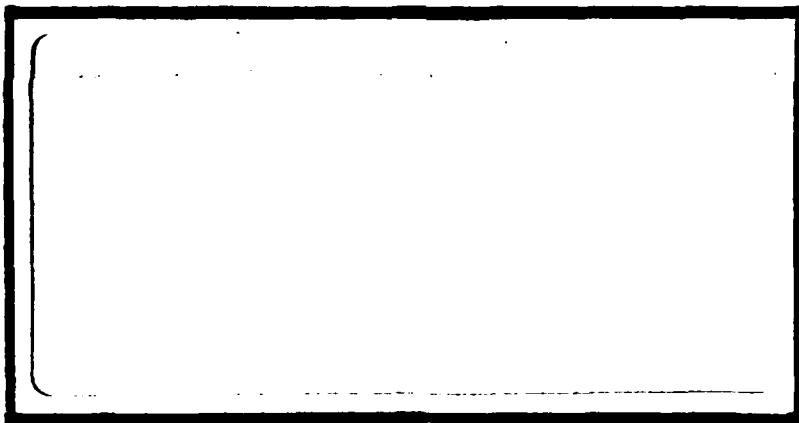
AD-A189 844 A METHODOLOGY BASED ON ANALYTICAL MODELING FOR THE  
DESIGN OF PARALLEL RND. (U) AIR FORCE INST OF TECH  
WRIGHT-PATTERSON AFB OH SCHOOL OF ENGI.. T G KEARNS  
UNCLASSIFIED DEC 87 AFIT/DS/ENG/87-1 F/G 12/6 NL





DTIC FILE COPY

AD-A189 844



DTIC  
ELECTED  
MAR 07 1988  
**S H D**

DEPARTMENT OF THE AIR FORCE  
AIR UNIVERSITY  
**AIR FORCE INSTITUTE OF TECHNOLOGY**

Wright-Patterson Air Force Base, Ohio

DISTRIBUTION STATEMENT A

Approved for public release  
Distribution Unlimited

88 3 1 190

(1)

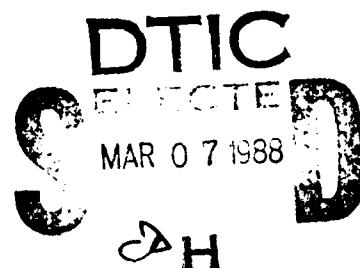
AFIT/DS/ENG/87-1

A METHODOLOGY, BASED ON ANALYTICAL  
MODELING, FOR THE DESIGN OF PARALLEL  
AND DISTRIBUTED ARCHITECTURES FOR  
RELATIONAL DATABASE QUERY PROCESSORS

DISSERTATION

Timothy G. Kearns  
Captain, USAF

AFIT/DS/ENG/87-1



Approved for public release; distribution unlimited

AFIT/DS/ENG/87-1

A METHODOLOGY, BASED ON ANALYTICAL MODELING.  
FOR THE DESIGN OF PARALLEL AND DISTRIBUTED  
ARCHITECTURES FOR RELATIONAL DATABASE QUERY  
PROCESSORS

DISSERTATION

Presented to the Faculty of the School of Engineering  
of the Air Force Institute of Technology  
Air University  
In Partial Fulfillment of the  
Requirements for the Degree of  
Doctor of Philosophy

Timothy G. Kearns, B.S., M.S.  
Captain, USAF

December 1987

Approved for public release: distribution unlimited

A METHODOLOGY, BASED ON ANALYTICAL MODELING.  
FOR THE DESIGN OF PARALLEL AND DISTRIBUTED  
ARCHITECTURES FOR RELATIONAL DATABASE QUERY  
PROCESSORS

Timothy G. Kearns, B.S., M.S.

Captain, USAF

Approved:

<u>Thomas C. Hartrum</u>	<u>11 Dec 87</u>
Thomas C. Hartrum, Chairman	
<u>Gary B. Lamont</u>	<u>14 Dec 87</u>
Gary B. Lamont	
<u>Henry B. Potocny</u>	<u>Dec. 11, 1987</u>
Henry B. Potocny	
<u>Nathaniel J. Davis, IV</u>	<u>12/11/87</u>
Nathaniel J. Davis, IV, Capt, USA	
<u>Mark A. Roth</u>	<u>11 Dec 87</u>
Mark A. Roth, Capt, USAF	

Accepted:

J. S. Przemieniecki 14 Dec. 87  
J. S. Przemieniecki  
Dean, School of Engineering

## *Preface*

The purpose of the research was to provide the capability to design a database machine that provides faster data retrievals by utilizing parallel processing. The limited results show the potential of using the design capabilities presented here to provide a physical design and implementation of a multiprocessor query processor.

This research would not have been possible without assistance from many people. To all of these people, I am greatly indebted and express my appreciation. In particular, I must thank Dr. Thomas C. Hartrum, my advisor. Without the assistance and guidance of Dr. Hartrum and his occasional boost to the rear-end to keep working, this project would have never been completed. Thank You!

I also wish to thank the members of my research committee, Dr. Lamont, Dr. Potoczny, Captain Davis, and Captain Roth, the dean's representative, for their guidance and assistance, especially in the final days of completing this project. Also, I would like to recognize my office mates of the last three years, Major Jim McManama, Capt Ed McCall, Capt Randy Paschall, Capt Paul Bailor, Capt Stu Sheldon, and Capt Chuck Matson, for providing a sympathetic ear and the encouragement to keep working even in the tough times. Finally, I wish to thank my wife, Pat, and my sons, Mark and Timmy, for their understanding and assistance, especially in these last months when spending evenings at school was the "thing to do".

Timothy G. Kearns



Accession For	
NTIS	Chart
DTIC TAB	
Unarranged	
Justified	
Revised	
Distr. by	
Approved by	
Dist. by	
A-1	

## *Table of Contents*

	Page
Preface . . . . .	iii
Table of Contents . . . . .	iv
List of Figures . . . . .	x
List of Tables . . . . .	xiv
List of Models . . . . .	xv
Abstract . . . . .	xxii
I. Introduction . . . . .	1
1.1 Introduction . . . . .	1
1.2 Problem . . . . .	4
1.3 Scope and Assumptions . . . . .	5
1.4 Background . . . . .	6
1.4.1 Back-end Conventional System. . . . .	7
1.4.2 Intelligent Controllers. . . . .	9
1.4.3 Multiprocessor Systems. . . . .	11
1.4.4 Special Hardware Systems. . . . .	15
1.4.5 Commercial Database Machines. . . . .	15
1.5 Approach . . . . .	17
1.6 Organization . . . . .	20

	Page
II. The Feasibility of Relational Operators with Partitioned Relations . . . . .	21
2.1 Data Partitioning . . . . .	21
2.2 Horizontal Partitioning . . . . .	22
2.3 Vertical Partitioning . . . . .	23
2.4 Relational Operators . . . . .	24
2.4.1 Additional Properties. . . . .	28
2.5 Select . . . . .	29
2.5.1 Horizontal Fragment Selects. . . . .	29
2.5.2 Vertical Fragment Selects. . . . .	31
2.6 Projection . . . . .	32
2.6.1 Horizontal Fragmentation Project. . . . .	33
2.6.2 Vertical Fragmentation Project. . . . .	34
2.7 Cartesian Product . . . . .	34
2.7.1 Horizontal Fragment Product. . . . .	35
2.7.2 Vertical Fragment Product. . . . .	36
2.8 Join . . . . .	37
2.8.1 Horizontal Fragment Join. . . . .	38
2.8.2 Vertical Fragment Join. . . . .	39
2.9 Union . . . . .	40
2.9.1 Horizontal Fragment Union. . . . .	41
2.9.2 Vertical Fragment Union. . . . .	41
2.10 Difference . . . . .	42
2.10.1 Horizontal Fragment Difference. . . . .	42
2.10.2 Vertical Fragment Difference. . . . .	45
2.11 Multi-way Operations . . . . .	45
2.12 Conclusions . . . . .	48

	Page
III. Modeling the Performance of the Select Operator . . . . .	50
3.1 Data Storage Structures . . . . .	51
3.1.1 Single Data Storage Structures. . . . .	52
3.1.2 Multiple Data Storage Structures. . . . .	53
3.2 Select Performance Models . . . . .	54
3.2.1 Case 1. Select - Single Processor-Single Disk. . . . .	55
3.2.2 Case 2. Select - Single Processor-Multiple Disks. . . . .	67
3.2.3 Case 3. Select - Multiple Processors-Single Disk. . . . .	77
3.2.4 Case 4. Select - Multiple Processors-Multiple Disks. . . . .	83
3.2.5 Summary. . . . .	97
IV. Modeling the Performance of the Project Operator . . . . .	100
4.1 Case 1. Project - Single Processor-Single Disk . . . . .	101
4.2 Case 2. Project - Single Processor-Multiple Disks . . . . .	105
4.3 Case 3. Project - Multiple Processors-Single Disk . . . . .	107
4.4 Case 4. Project - Multiple Processors-Multiple Disks. . . . .	112
4.5 Summary . . . . .	116
V. Modeling the Performance of the Join Operator . . . . .	117
5.1 Case 1. Join - Single Processor-Single Disk . . . . .	118
5.1.1 Nested-Loop. . . . .	119
5.1.2 Sort-Merge. . . . .	121
5.1.3 Indexing. . . . .	126
5.2 Case 2. Join - Single Processor-Multiple Disks . . . . .	142
5.2.1 Nested-Loop. . . . .	142
5.2.2 Sort-Merge. . . . .	143
5.2.3 Indexing. . . . .	145
5.3 Case 3. Join - Multiple Processors-Single Disk . . . . .	153

	Page
5.3.1 Nested-Loop. . . . .	153
5.3.2 Sort-Merge. . . . .	154
5.3.3 Indexing. . . . .	157
5.4 Case 4. Multiple Processors-Multiple Disks . . . . .	157
5.4.1 Nested-Loop. . . . .	159
5.4.2 Sort-Merge. . . . .	160
5.4.3 Bucket Join or Hash Join. . . . .	167
5.5 Summary . . . . .	170
 VI. Update Performance Effects . . . . .	 172
6.1 Inserting a new tuple. . . . .	172
6.1.1 Single Processor - Single Disk Insertion. . . . .	173
6.1.2 Single Processor - Multiple Disk Insertion. . . . .	176
6.1.3 Multiple Processor - Single Disk Insertion. . . . .	178
6.1.4 Multiple Processor - Multiple Disk Insertion. . . . .	179
6.2 Deleting a tuple. . . . .	181
6.2.1 Single Processor - Single Disk Deletion. . . . .	182
6.2.2 Single Processor - Multiple Disk Deletion. . . . .	183
6.2.3 Multiple Processor - Single Disk Deletion. . . . .	184
6.2.4 Multiple Processor - Multiple Disk Deletion. . . . .	186
6.3 Modifying a tuple . . . . .	188
6.3.1 Single Processor - Multiple Disk Modification. . . . .	188
6.3.2 Single Processor - Multiple Disk Modification. . . . .	191
6.3.3 Multiple Processor - Single Disk Modification. . . . .	193
6.3.4 Multiple Processor - Multiple Disk Modification. . . . .	195
6.4 Summary . . . . .	198

	Page
VII. Single Query Step Model Results . . . . .	199
7.1 Select . . . . .	200
7.2 Project . . . . .	202
7.3 Join . . . . .	205
7.4 Bucket Join . . . . .	208
7.5 Update . . . . .	215
7.5.1 Insertion . . . . .	215
7.5.2 Deletion . . . . .	218
7.5.3 Modification . . . . .	218
7.6 Performance Conclusions . . . . .	219
VIII. Multi-Step Query Performance . . . . .	223
8.1 Query Tree . . . . .	223
8.2 Combined Operators . . . . .	228
8.2.1 The Sel-Proj Operation. . . . .	228
8.2.2 Other Combined Operations. . . . .	229
8.3 General Query Tree Form . . . . .	230
8.3.1 Initial Nodes in Normal Form Query. . . . .	231
8.3.2 Binary Nodes in Normal Form Query. . . . .	232
8.4 Performance of Normal Form Queries . . . . .	236
8.5 Modeling a Multi-Step Query . . . . .	239
8.5.1 Multi-Step Query Processing using Consecutive Relation Retrieval. . . . .	242
8.5.2 Multi-Step Query Processing using Concurrent Relation Retrieval. . . . .	250
8.5.3 Multi-Step Query Processing using Consecutive Relation Retrieval with Dedicated Join Processors. .	252
8.5.4 Multi-Step Query Processing using Concurrent Relation Retrieval with Dedicated Join Processors. .	254

	Page
8.5.5 Multi-Step Modeling Results. . . . .	255
8.6 Extension of Model to Multi-Binary Node Queries . . . . .	257
8.7 Control of Resources and Task Allocation . . . . .	263
8.8 Summary of Combined Step Effects . . . . .	269
 IX. Database Machine Architecture . . . . .	 272
9.1 Database Machine Architecture . . . . .	274
9.1.1 Retrieval Layer. . . . .	274
9.1.2 Processing Layer. . . . .	280
9.2 A Database Machine Design . . . . .	283
9.2.1 Retrieval Stage. . . . .	285
9.2.2 Retrieval-Processing Stage Interconnection. . . . .	286
9.2.3 Processing Stage. . . . .	287
9.2.4 Controller. . . . .	291
 X. Conclusions and Recommendations . . . . .	 293
10.1 Conclusions . . . . .	293
10.2 Recommendations . . . . .	296
 A. Multi-Step Query Results . . . . .	 298
B. Multi-Binary Node Performance Results . . . . .	308
C. Benchmark Performance Comparison . . . . .	314
 Bibliography . . . . .	 318
Vita . . . . .	325

## *List of Figures*

Figure	Page
1. Back-End Conventional System . . . . .	7
2. Intelligent Disk Controller . . . . .	8
3. Multiprocessor System . . . . .	8
4. Processor-Per-Track . . . . .	10
5. Processor-Per-Head . . . . .	11
6. Processor-Per-Disk . . . . .	12
7. MDBS Architecture . . . . .	13
8. AFIT MPOA Architecture . . . . .	13
9. DIRECT Architecture . . . . .	14
10. Teradata Ynet Architecture . . . . .	16
11. Horizontal Partitioning . . . . .	22
12. Vertical Partitioning . . . . .	24
13. Examples of the Select Operation . . . . .	30
14. Examples of Selection with Vertical Fragments . . . . .	32
15. Examples of the Projection Operation . . . . .	33
16. The Product Operation . . . . .	35
17. The Join (Equi-Join) Operation . . . . .	38
18. The Union Operation . . . . .	41
19. Difference operation . . . . .	42
20. Difference with Horizontal Fragments and a Relation . . . . .	44
21. Difference with Horizontal Fragments . . . . .	46
22. Multiple Processor-Disk Performance for the FT Select Operation . . . . .	202
23. Multiple Processor-Multiple Disk Performance for the MT Select Operation . . . . .	203
24. Multiple Processor-Multiple Disk Performance for the Project Operation	204

Figure		Page
25.	The Performance Effect of Increasing Resources on the Project Operation . . . . .	204
26.	Join Performance using a Single Processor and Multiple Disk. . . . .	206
27.	Join Performance using Multiple Disks and Multiple Processors. . . . .	207
28.	Join Performance,including the Bucket Join, using Multiple Disks and Multiple Processors. . . . .	207
29.	Join Performance Effects of Increasing Processors and Secondary Storage Using Different Algorithms . . . . .	208
30.	Join Performance Effects of Increasing Processors and Secondary Storage Varying the Input Size . . . . .	209
31.	Join Performance Effects of Decreasing Access Speed of Secondary Storage. . . . .	211
32.	Join Performance Effects of Communication Speed. . . . .	212
33.	Join Performance Effects of Join Selectivity Factor. . . . .	212
34.	Join Performance Effects of Increased Processor Memory. . . . .	214
35.	The Performance of Inserting a Tuple . . . . .	217
36.	The Performance of Deleting a Tuple . . . . .	218
37.	The Performance of Modifying a Tuple . . . . .	219
38.	A Simple Query Tree . . . . .	224
39.	Directed Graph in the Query Tree . . . . .	225
40.	Example Query . . . . .	227
41.	Sel-Proj Operation versus Select and Project using Multiple Processors-Multiple Disks. . . . .	229
42.	General Normal Form Query Tree . . . . .	231
43.	Example Query . . . . .	232
44.	Normal Form Query Tree with Duplicate Removal . . . . .	235
45.	Process Overlapping of Multi-Step Query . . . . .	243
46.	Process Overlapping of Multi-Step Query with Overlapping Relation Retrievals . . . . .	244

Figure	Page
47. Logical View of Case 3 Multi-Step Query Processing . . . . .	253
48. Multi-Step Performance using Hash/Join . . . . .	257
49. Multi-Step Performance using Separate Hash and Join . . . . .	258
50. Performance with Selectivity Factor = 1, Projectivity Factor = 100% using 5 Retrieval Processors - 1 . . . . .	258
51. Performance with Selectivity Factor = 1, Projectivity Factor = 100% using 5 Retrieval Processors - 2 . . . . .	259
52. Processing Overlap Model . . . . .	260
53. Processing Overlap Model with Retrieval Processor Constraint . . .	262
54. The Logical Database Machine Environment . . . . .	275
55. General Normal Form Query Tree . . . . .	276
56. Logical Database Machine Architecture . . . . .	276
57. Logical Data Flow of Database Machine . . . . .	277
58. Modified Logical Database Machine Architecture . . . . .	281
59. Logical Database Machine Architecture . . . . .	282
60. A Database Architecture . . . . .	284
61. Logical Operation of Modified Hash/Join Process . . . . .	290
62. Multi-Step Performance using Hash/Join . . . . .	299
63. Multi-Step Performance using Separate Hash and Join - 1 . . . . .	299
64. Multi-Step Performance using Separate Hash and Join - 2 . . . . .	300
65. Multi-Step Performance using Separate Hash and Join - 3 . . . . .	300
66. Multi-Step Performance using Separate Hash and Join - 4 . . . . .	301
67. Multi-Step Performance using Separate Hash and Join - 5 . . . . .	301
68. Performance with Selectivity Factor = 1, Projectivity Factor = 100% using 10 Retrieval Processors - 1 . . . . .	302
69. Performance with Selectivity Factor = 1, Projectivity Factor = 100% using 10 Retrieval Processors - 2 . . . . .	302
70. Performance with Selectivity Factor = 1, Projectivity Factor = 100% using 15 Retrieval Processors - 1 . . . . .	303

Figure		Page
71.	Performance with Selectivity Factor = 1, Projectivity Factor = 100% using 15 Retrieval Processors - 2 . . . . .	303
72.	Performance with Selectivity Factor = .5, Projectivity Factor = 100% using 15 Retrieval Processors - 1 . . . . .	304
73.	Performance with Selectivity Factor = .5, Projectivity Factor = 100% using 15 Retrieval Processors - 2 . . . . .	304
74.	Performance with Selectivity Factor = .5, Projectivity Factor = 50% using 15 Retrieval Processors - 1 . . . . .	305
75.	Performance with Selectivity Factor = .5, Projectivity Factor = 50% using 15 Retrieval Processors - 2 . . . . .	305
76.	Performance with Selectivity Factor = .1, Projectivity Factor = 50% using 15 Retrieval Processors - 1 . . . . .	306
77.	Performance with Selectivity Factor = .1, Projectivity Factor = 50% using 15 Retrieval Processors - 2 . . . . .	306
78.	Performance with Selectivity Factor = .1, Projectivity Factor = 10% using 15 Retrieval Processors - 1 . . . . .	307
79.	Performance with Selectivity Factor = .1, Projectivity Factor = 10% using 15 Retrieval Processors - 2 . . . . .	307
80.	Multi-Binary Node Query Model . . . . .	309
81.	Multi-Binary Node Performance Graph - 1 . . . . .	311
82.	Multi-Binary Node Performance Graph - 2 . . . . .	312
83.	Multi-Binary Node Performance Graph - 3 . . . . .	312
84.	Multi-Binary Node Performance Graph - 4 . . . . .	313
85.	Multi-Binary Node Performance Graph - 5 . . . . .	313
86.	Benchmark Results . . . . .	316

### *List of Tables*

Table	Page
1. Feasibility of Operators with Fragments . . . . .	48
2. Performance equation parameters . . . . .	56
3. Performance Model Workload Parameters . . . . .	200
4. Summary of Performance Results . . . . .	222

### *List of Models*

Model	Page
S - 1 .....	57
S - 2 .....	57
S - 3 .....	58
S - 4 .....	58
S - 5 .....	59
S - 6 .....	59
S - 7 .....	61
S - 8 .....	61
S - 9 .....	62
S - 10 .....	63
S - 11 .....	63
S - 12 .....	64
S - 13 .....	65
S - 14 .....	65
S - 15 .....	66
S - 16 .....	66
S - 17 .....	66
S - 18 .....	66
S - 19 .....	67
S - 20 .....	67
S - 21 .....	69
S - 22 .....	69
S - 23 .....	69
S - 24 .....	70
S - 25 .....	71
S - 26 .....	71
S - 27 .....	72
S - 28 .....	72
S - 29 .....	73
S - 30 .....	74
S - 31 .....	74

Model	Page
S - 32 .....	75
S - 33 .....	76
S - 34 .....	76
S - 35 .....	76
S - 36 .....	77
S - 37 .....	78
S - 38 .....	78
S - 39 .....	79
S - 40 .....	80
S - 41 .....	80
S - 42 .....	81
S - 43 .....	82
S - 44 .....	82
S - 45 .....	82
S - 46 .....	83
S - 47 .....	85
S - 48 .....	85
S - 49 .....	85
S - 50 .....	86
S - 51 .....	87
S - 52 .....	88
S - 53 .....	89
S - 54 .....	90
S - 55 .....	90
S - 56 .....	90
S - 57 .....	91
S - 58 .....	91
S - 59 .....	91
S - 60 .....	92
S - 61 .....	92
S - 62 .....	93
S - 63 .....	93
S - 64 .....	94
S - 65 .....	94
S - 66 .....	95

Model	Page
S - 67 .....	95
S - 68 .....	96
S - 69 .....	96
P - 1 .....	102
P - 2 .....	102
P - 3 .....	103
P - 4 .....	104
P - 5 .....	105
P - 6 .....	106
P - 7 .....	106
P - 8 .....	107
P - 9 .....	108
P - 10 .....	108
P - 11 .....	110
P - 12 .....	111
P - 13 .....	113
P - 14 .....	113
P - 15 .....	113
P - 16 .....	115
J - 1 .....	120
J - 2 .....	120
J - 3 .....	122
J - 4 .....	122
J - 5 .....	123
J - 6 .....	125
J - 7 .....	125
J - 8 .....	126
J - 9 .....	126
J - 10 .....	129
J - 11 .....	132
J - 12 .....	133
J - 13 .....	133
J - 14 .....	134
J - 15 .....	134
J - 16 .....	135

Model	Page
J - 17 .....	135
J - 18 .....	136
J - 19 .....	136
J - 20 .....	136
J - 21 .....	136
J - 22 .....	137
J - 23 .....	137
J - 24 .....	138
J - 25 .....	138
J - 26 .....	139
J - 27 .....	139
J - 28 .....	139
J - 29 .....	140
J - 30 .....	140
J - 31 .....	140
J - 32 .....	141
J - 33 .....	141
J - 34 .....	141
J - 35 .....	142
J - 36 .....	142
J - 37 .....	143
J - 38 .....	143
J - 39 .....	144
J - 40 .....	144
J - 41 .....	145
J - 42 .....	145
J - 43 .....	146
J - 44 .....	147
J - 45 .....	147
J - 46 .....	147
J - 47 .....	148
J - 48 .....	148
J - 49 .....	148
J - 50 .....	149
J - 51 .....	150

Model	Page
J - 52 .....	150
J - 53 .....	150
J - 54 .....	151
J - 55 .....	151
J - 56 .....	151
J - 57 .....	152
J - 58 .....	152
J - 59 .....	153
J - 60 .....	153
J - 61 .....	154
J - 62 .....	154
J - 63 .....	154
J - 64 .....	156
J - 65 .....	156
J - 66 .....	156
J - 67 .....	157
J - 68 .....	160
J - 69 .....	160
J - 70 .....	160
J - 71 .....	162
J - 72 .....	162
J - 73 .....	162
J - 74 .....	163
J - 75 .....	164
J - 76 .....	166
J - 77 .....	167
J - 78 .....	169
J - 79 .....	169
U - 1 .....	173
U - 2 .....	174
U - 3 .....	175
U - 4 .....	175
U - 5 .....	176
U - 6 .....	177
U - 7 .....	177

Model	Page
U - 8 .....	178
U - 9 .....	179
U - 10 .....	179
U - 11 .....	180
U - 12 .....	180
U - 13 .....	181
U - 14 .....	181
U - 15 .....	182
U - 16 .....	182
U - 17 .....	183
U - 18 .....	183
U - 19 .....	184
U - 20 .....	185
U - 21 .....	185
U - 22 .....	186
U - 23 .....	186
U - 24 .....	187
U - 25 .....	187
U - 26 .....	188
U - 27 .....	189
U - 28 .....	190
U - 29 .....	191
U - 30 .....	192
U - 31 .....	192
U - 32 .....	193
U - 33 .....	194
U - 34 .....	194
U - 35 .....	195
U - 36 .....	196
U - 37 .....	196
U - 38 .....	197
U - 39 .....	197
M - 1 .....	250
M - 2 .....	252
M - 3 .....	254

Model	Page
M - 4 .....	254

*Abstract*

The design of faster relational database query processors to improve the data retrieval capability of a database was the goal of this research. The emphasis was on evaluating the potential of parallel implementations to allow use of multiprocessing. First, the theoretical properties of applying relational operations to distributed data were considered to provide an underlying data distribution and parallel processing environment model. Next, analytical models were constructed to evaluate various implementations of the select, project, and join relational operations and the update operations of addition, deletion, and modification for a range of data structures and architectural configurations. To simulate the performance of the query processor for all cases, the individual operator models needed to be extended for complex queries consisting of several relational operations. A solution to modeling multi-step queries was the use of a general normal form to express a query. This normal form query tree used combined operations to express relational algebra equivalent queries in a standard form. This standard tree form was then used to construct analytical models for multi-step queries. These models provide the capability to simulate the potential of different forms of parallelism in solving complex queries. The analysis of results of the analytical models presents a logical design for a multiprocessor query processor. This logical query processor using multiple processors and employing parallelism illustrated the potential for an improved query processor when the analytical model results of complex queries were compared to a benchmark of some current database systems.

A METHODOLOGY, BASED ON ANALYTICAL MODELING,  
FOR THE DESIGN OF PARALLEL AND DISTRIBUTED  
ARCHITECTURES FOR RELATIONAL DATABASE QUERY  
PROCESSORS

*I. Introduction*

*1.1 Introduction*

The demand for efficient and timely management of data is growing stronger every day. The database management systems (DBMS) of the 70s were supposed to be the solution to the need for better data management facilities [79,19]. The DBMS provided a partial solution, however the widespread expansion of affordable computers has caused an increased expectation and demand in non-numeric processing [60]. The expectations of the data management facilities include managing larger data bases, easier manipulation of the data, and faster access to the data. The size of databases the DBMS needs to manage is growing [60]. Although the DBMS solution may provide better data sharing and easier methods of managing data, it still lacks the performance to find and retrieve data as quickly as needed for many applications using these large data bases. A possible near-future solution to the data management performance problem is the database machine.

Database machines provide the same database management tools as a conventional DBMS but provide improved performance. The improved performance of the database machine is due to several reasons [20,41]. First, since the database machine is a dedicated hardware system that is a back-end to the conventional host, there is some inherent parallelism [20]. The dedicated hardware allows the environment

of the DBMS to be tailored for the system demands. In the general purpose (share the machine with many other applications) computer system, the database management system must compete in the general environment for resources and services and in some cases the DBMS duplicates functions of the operating systems [76]. By dedicating the hardware to a single purpose, specialized hardware may be used. Eliminating the operating system of the computer and building a unique database operating system may also increase performance [74]. Further discussion of possible performance improvements in database machines follows in a later section. Although performance is the main advantage of a database machine, there are several other attractive features of the database machine.

The database machine has all of the advantages associated with a conventional DBMS plus the following additional improvements [20,51,52]:

**Reliability** The conventional software database management system is a large complex software system. Since the conventional system is so large and complex, it is difficult to verify and may be prone to failure. The database machine provides the capability to implement in hardware some normally software functions (i.e., sorts and merges) [51] to possibly reduce the size and complexity of the software. Thus, the reliability may be improved.

**Security** The database machine is a back-end machine of the host. Therefore, the host filters the requests that are sent to the database machine creating another layer of security [20]. Also, the database may improve security checking since it is dedicated to only data management and not supporting many different applications.

**Host Independence and Modularity** The database machine is a standalone system that runs separate from the host. A physical channel(s) connection to the host is all that is required to pass requests and data. This allows more database machine modules to be added to the system as needed.

**Multiple Hosts and Database Sharing** The database machine may be accessed by several different, dissimilar hosts. This provides the ability for databases to be shared by many different users and applications, thus reducing the need for expensive replication of data.

**Cost/Performance** The database machine reduces the demands on the host I/O system by providing its own I/O system. This allows the host to provide better support for its applications. Also, since the database machine supports only a specific task, its disk and I/O hardware may be more efficiently utilized.

The database machine provides the potential for faster, more secure, and reliable data processing than the conventional DBMS [20], but the current database machines do not present the definitive solution to database management and still have areas where further advancements may be possible. First, most of the database machine researchers define an architecture or specific hardware device first, then develop algorithms or methods to complete the system, causing a less than optimum solution for the total system. Next, although the relational data model is the data model of choice at the current time, other data models or storage configurations of data must not be forgotten. And last, the future seems to require more processing power, and at this time the only feasible way to do this (due to the high cost and limited availability of specialized hardware) is with parallel processing.

Multiprocessing or parallel processing is already being explored in research of database machines [21,30,31,43,56,62,66,70]. However, the architectures using multiprocessing at the current time all have expansion limitations [14]. Ideally, a database machine could be expanded by just adding modules (processors) and more storage devices (disks). Modular expansion capability requires a flexible, expandable control structure, generic processors and some form of processor communication network. The largest problem of networking is controlling the processors and distributing data so that all the processors can be efficiently utilized (i.e., each processor handles an equal amount of data without using excessive time to distribute the data or telling

the processors what to do). This will require some form of decentralized control of the processors, effective distribution of the data for processing [14], and algorithms designed for the environment.

If the processor network can be efficiently controlled, the data still must be stored on secondary storage to be efficiently retrieved. Magnetic and laser disk technology have provided disks that can store very large amounts of data. But, in what format should the data be stored on disk and how should the data be distributed on the disk drives? These are both questions that have been examined [27] but no definitive solution has been found for all cases.

The last area of neglected research in database machine theory is "data updating". The goal of current database machines and database machine research is optimizing data retrievals [60]. This goal is appropriate since the demand of the user community has been for faster retrievals. But, in optimizing the retrievals little consideration is given to how including concurrent data updating affects the performance of the data retrievals. For many new applications, efficient updating may be as important as the retrievals. One example of this is the Strategic Defense Initiative's need to have inputs from thousands of sensor and detector devices to be used in making decisions. This data must be shared with many processes and still stored for later decisions. This requires efficient updating of the data as well as providing immediate retrieval capabilities for numerous users.

## 1.2 *Problem*

The focus of this research was to develop a methodology and tools to design a multiprocessor database machine that could provide improved performance by decreasing the time required to execute queries. The methodology development examined the individual components of the problem and integrated the results to provide the capability to design a multiprocessor database machine. The conclusion of this development is a logical architecture and set of analytical models to guide

the development of a database query processor.

The individual relational operators were theoretically examined and analyzed to determine the "optimal" method of utilizing parallel processing for relational operators in a multiprocessor system, the most efficient implementation (in terms of time required to complete the operation) of data retrieval and data update algorithms in a multiprocessor environment, and the effects of the data structure on the performance of retrieval and updates. The results and analysis of these objectives was extended to provide the ability to evaluate the performance of complex queries consisting of several retrieval steps. This provides the basis for a logical database machine architecture and a methodology for designing a database machine.

### *1.3 Scope and Assumptions*

Theoretical complications and practical limitations necessitate the imposition of some assumptions and conditions on the problem.

1. This research assumed an environment that consisted of multiple processors, where each processor can operate independently and each processor may communicate with other processors. Each processing unit will be a generic type processing unit containing a processor, memory, and the ability to execute a stored program.
2. The database machine is assumed to be capable of handling any size database (to some reasonable maximum size). This removes the restriction that the database must be small enough to fit in main memory [26].
3. The focus of the research was improving the performance of data retrievals and data updates. Therefore, backup, checkpoints, and recovery were not considered in the design of the system. These are essential features of a production system but are peripheral to the design of the database machine for improved query processing.

4. The performance improvement strived for by this research concentrated on improvement through the use of parallel processing and efficient utilization of resources. Optimization by reordering query steps and other optimization techniques of this nature [73,82] were not considered as critical elements in this research. However, any design should be modular enough to allow incorporating these techniques at a later time.
5. The performance of any database machine design is assumed to be overwhelmed by volume of data or transactions at some point. This causes the need for modular expansion of the system. Since modular expansion is not always possible, the ability to easily expand the capability was a critical design factor in this database machine research.
6. The design of the database machine did not consider possible limitations of currently available multiprocessor systems or the availability of multiprocessor systems.
7. The only non-procedural query languages currently used are relational calculus and relational algebra. Since relational algebra and relational calculus are equivalent [79], the non-procedural retrievals are considered to be relational operators.

#### *1.4 Background*

The term "database machine" does not mean a specific type of hardware system for implementing a database management system. Instead this term is applied to a range of ideas and methods to improve the performance of a database management system. The implementations discussed here are grouped into the following areas: the conventional back-end system, intelligent disk controllers and data filtering, multiprocessor systems, and specialized hardware systems. See Figures 1- 3, on the following pages, for simple block diagrams of the various configurations. Other

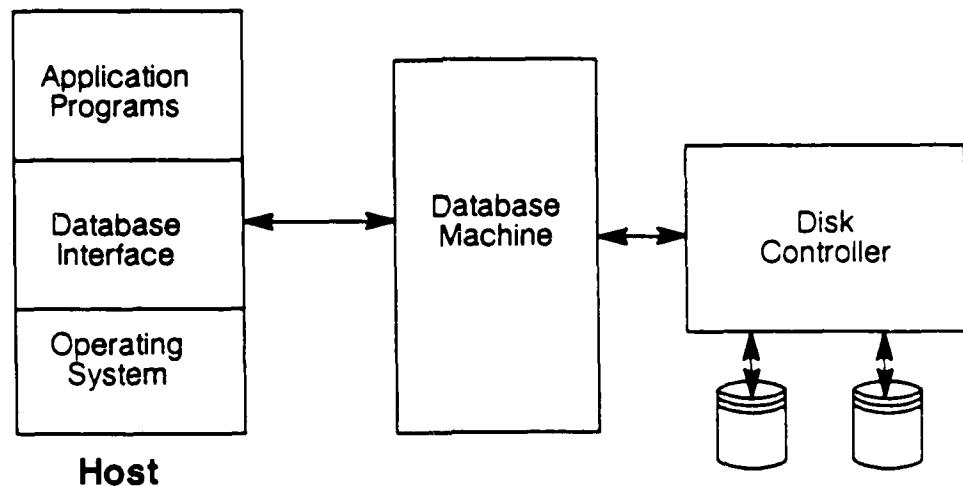


Figure 1. Back-End Conventional System

classification schemes may separate implementations by software oriented versus hardware oriented or specific types of hardware [11,20,25,33,43,65].

*1.4.1 Back-end Conventional System.* The conventional back-end system is the easiest system to implement. It consists of a general purpose computer system running only a database management system as a back-end machine to the host processor. The host processor runs the applications that need data from the databases. The host sends the request for data to the back-end machine which processes the request and returns the response.

The conventional back-end system just duplicates the general purpose type hardware of the host, but by providing a dedicated environment, better performance should be realized. This type of back-end system illustrates an important consideration when downloading work to another processor to try to get an improvement in performance: communication overhead. Every time a task is downloaded it requires at least two messages between the systems. If the overhead of the message traffic is

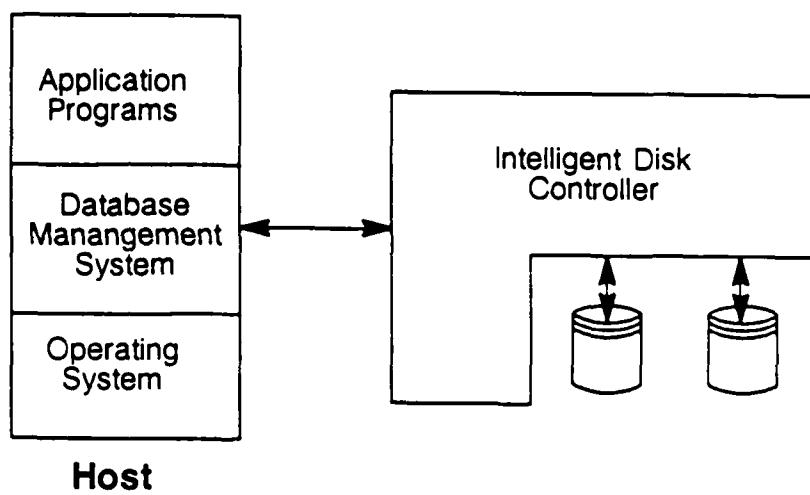


Figure 2. Intelligent Disk Controller

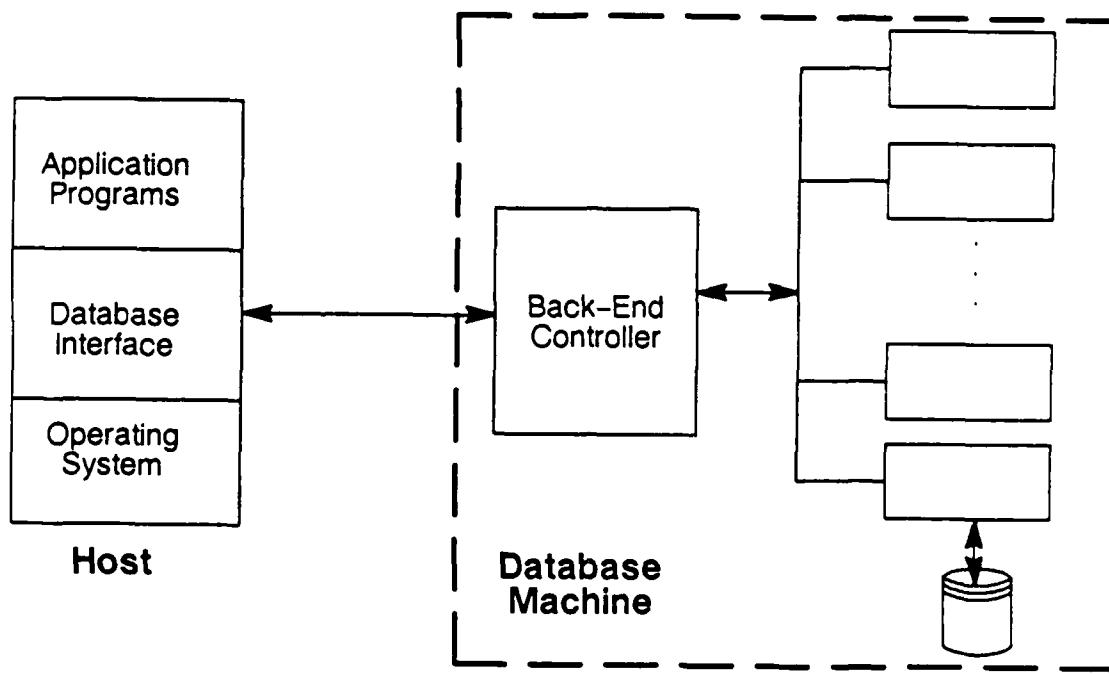


Figure 3. Multiprocessor System

more time expensive than processing the task on the host system. then downloading is not time efficient. Date [20] provides an excellent illustration of this point. This communication is the main reason that all the current investigations of database machines provide support only for nonprocedural query processing, like the relational data model.

*1.4.2 Intelligent Controllers.* The intelligent controller category of database machines consists of architectures that improve the performance of database management by improving the data retrieval from secondary storage devices. The idea of intelligent controllers is to move some of the processing logic for determining the data needed from the processor to the memory device or controller of the memory device. This improves performance of the processor by providing it with less data to manipulate. This reduces the communication overhead on the bus, reducing the potential for a bottleneck of the system.

The concept of intelligent controllers applies to magnetic disk technology, but many of the intelligent controller concepts were developed with the idea that other storage technologies, such as magnetic bubble memory and CCD, would eventually replace the magnetic disk. The other storage technologies eliminated the read/write head of disk technology, making them more appropriate for the intelligent controller concepts. Instead, at the current time, disk technology (magnetic and laser) has made the great advances in reducing the cost per byte of storage and the amount of storage space available on a device. Therefore, software improvements may be more important than intelligent controllers. The intelligent controller concepts may be divided into the following groups: associative disks, processor-per- head, and filters.

The first type of intelligent controller is an associative disk or processor-per-track device [29]. Figure 4 provides a simple diagram of a processor-per-track system. The concept of the processor-per-track device is to process selection operations “on the fly” [25]. This concept of providing processing on every cell or track is also

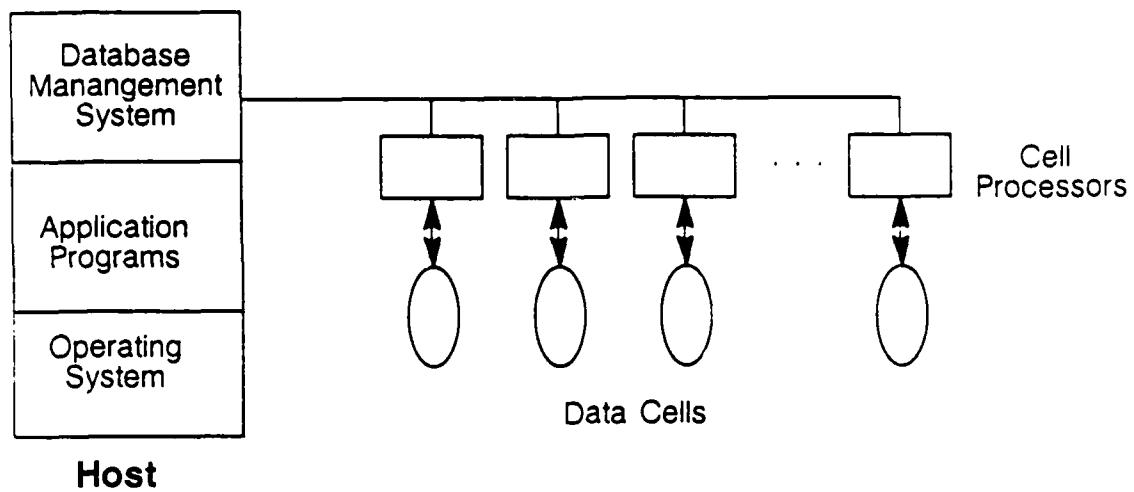


Figure 4. Processor-Per-Track

referred to as cellular-logic [77]. CASSM [50] and RAP [68,69] are examples of systems designed using cellular-logic.

Cellular-logic provides the capability to scan the set of data in one revolution of the storage device. The processing logic scans the data at each track, selects the appropriate data, and places the selected data in an output buffer.

The next type of controller, the processor-per-head, provides the same type of processing as the processor-per-track except on a more limited basis (see Figure 5). Processor-per-head devices have a moving head that may move from track-to-track. The head reads the data from the disk and places it in an input buffer for the processor for the selection logic to be applied. DBC was designed with this type device [4,43].

Filtering, or a processor-per-disk, is the last type of controller to be examined (see Figure 6). This method utilizes a standard disk drive and processor. All data from the disk is provided to the processor where selection logic selects the desired

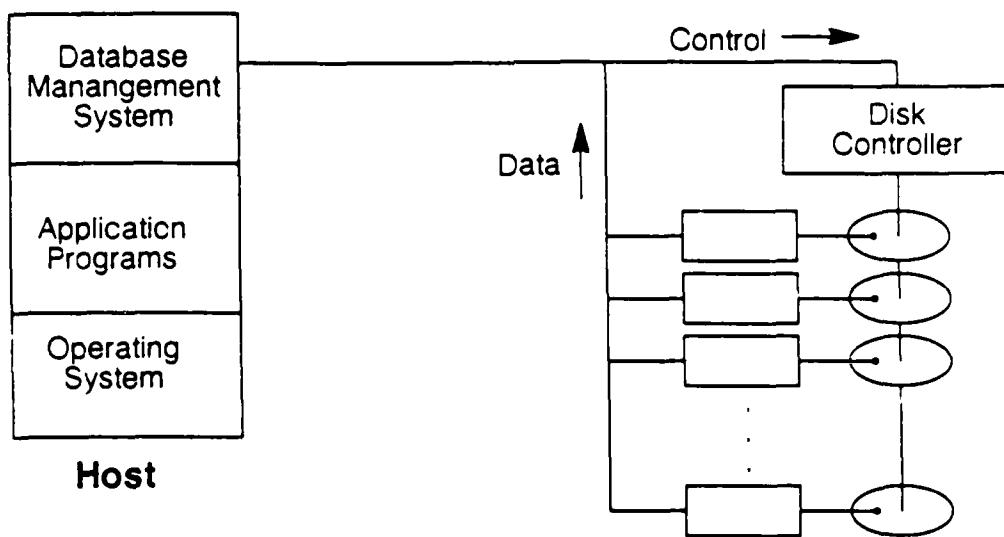


Figure 5. Processor-Per-Head

data. Therefore, the processor acts as a filter on the data before the data enters into the actual database management system. VERSO [3] and SABRE [32] are systems that incorporate filtering with other architecture features. With the advent of cheap disk technology and VLSI design of the filter, this could become an add-on feature for many conventional systems [72].

The performance of the intelligent controllers is very good for the tasks they are designed to do [25,33]. But, because of the specialized nature of the intelligent controller, they may not perform other tasks as well. This illustrates the fact that most architectures are developed before the complete implementation of the system is considered. Therefore, the software algorithms to implement many operations have to work around the hardware architecture, not work with it.

*1.4.3 Multiprocessor Systems.* Increasing the throughput of the system by using some form of parallel processing is the concept of multiprocessor systems. Multiprocessor systems include many different hardware configurations. Figures 7,

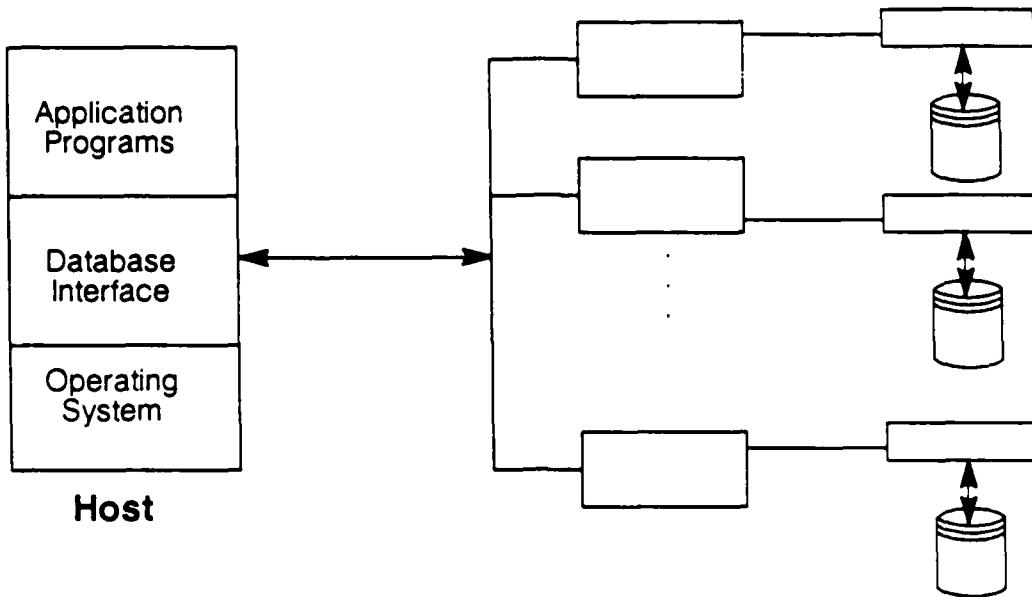


Figure 6. Processor-Per-Disk

8 and 9 provide some examples of different multiprocessor systems configurations. Some provide cache between the disk and processors [8] to allow faster data access. Others use the multiprocessing almost as disk filters [34.42].

The architecture of the multiprocessor systems must provide some means of providing communication and data transfer among processors. This communication capability is normally implemented using a bus to connect all the processors or using some form of a point-to-point network structure [67]. The different processor interconnection schemes provide different capabilities and opportunities to distribute data to the various processors for the best utilization of resources and to provide the most effective parallel processing. The following descriptions will examine some of the multiprocessor architectures and the associated features of each.

The Multibackend Database System (MDBS) [34.42] uses a processor for each disk of the system plus a single processor to control all of the disk processors (see Figure 7). When a query enters the system, the controller directs the processors on the disk to retrieve the data. Each disk processor then retrieves its portion of the

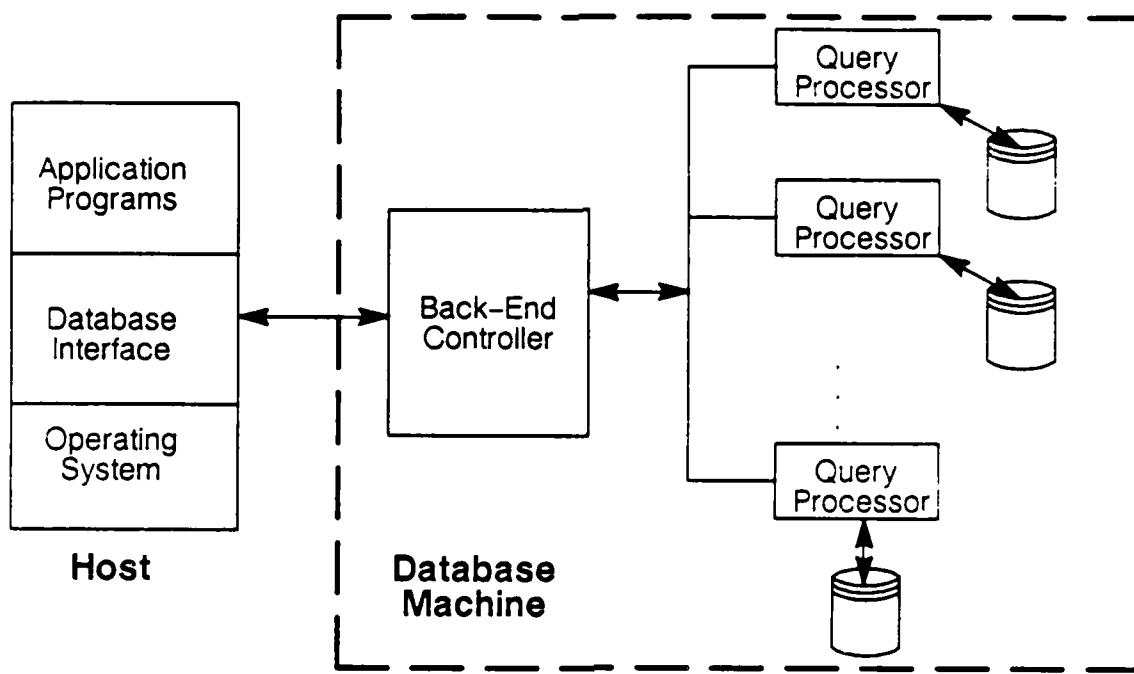


Figure 7. MDBS Architecture

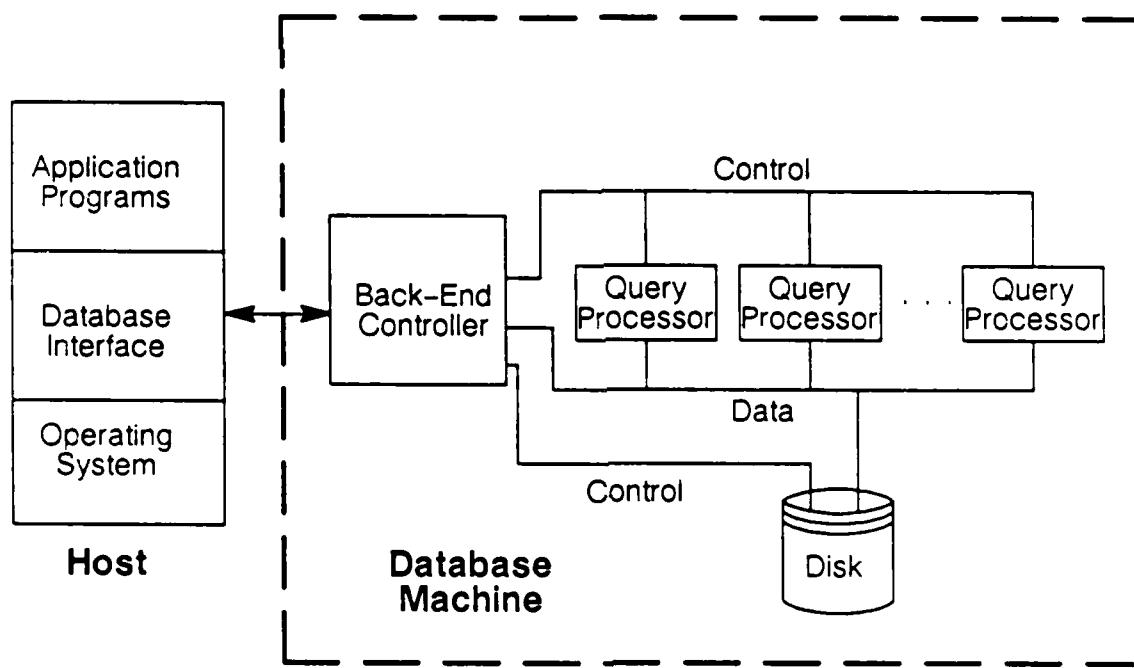


Figure 8. AFIT MPOA Architecture

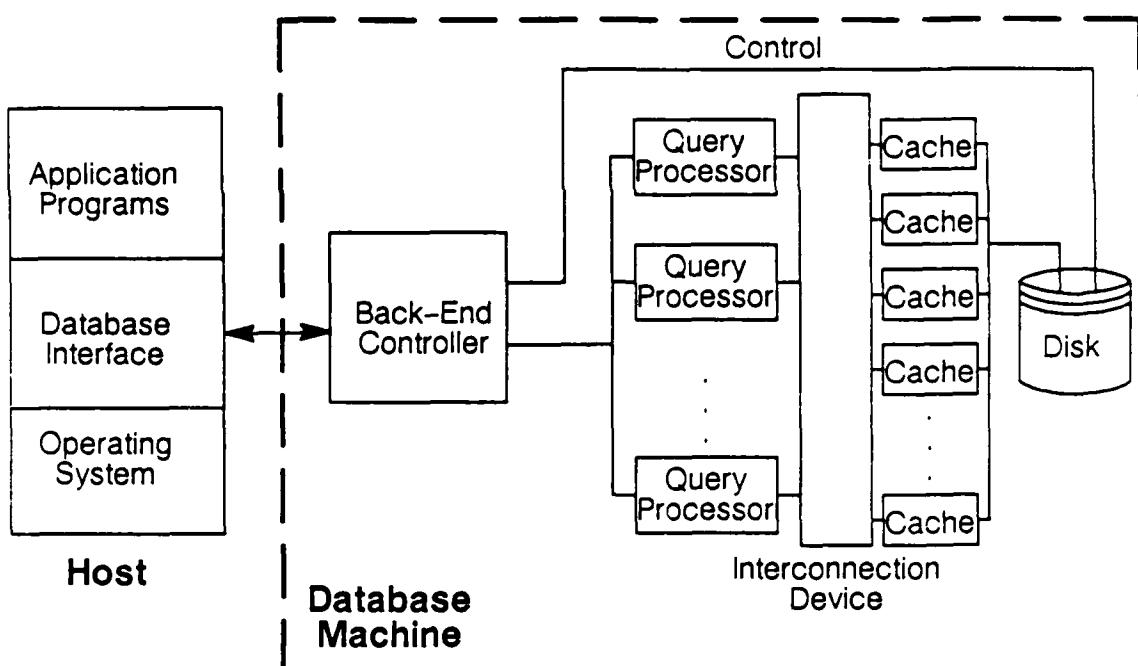


Figure 9. DIRECT Architecture

data from its disk and processes it. This makes this system seem to be a SIMD type architecture. The gain of this type of system as claimed by the developers is almost a linear decrease in query processing time when additional disk processors and disks are added and the data redistributed.

A more conventional use of multiprocessors is to employ a MIMD type architecture, where each processor may operate independently. There are many different configurations of database machines of MIMD type architecture [15,12,23,30,31,56, 62,66,67,70,81] but DIRECT (see Figure 9) is probably the most recognized. DIRECT [15,12,21,23] uses the multiprocessors in a MIMD type architecture with a single control processor. DIRECT also uses cache with an interconnection device to allow sharing of disk data among processors. The combination of the data sharing and processor control structure of DIRECT makes it very efficient for many relational query operations [25,33].

The controlling of the processors and sharing data are two of the main issues in a MIMD type architecture. Boral and DeWitt use the concept of data-flow [12] and

dynamic allocation of tasks to processors [15] to provide efficient implementation of query algorithms in DIRECT. Others also address the controlling of processors and how to optimize query execution with multiple processors [31,62,64,66,81]. The combination of dynamic allocation of tasks, load-balancing of the processing [66], and data-flow (or process-flow) concepts [15,62] provide areas of potential gain using different MIMD type structures such as the hypercube type network computer structures.

*1.4.4 Special Hardware Systems.* This category is added to the types of database machines because the advances in VLSI technology have made it possible to build specialized processors. These specialized processors may provide sorting functions, special join functions [53], or aggregate functions. By using the special hardware in the development of the query algorithms, specialized query processors may be constructed [38,37,44,47,48,58].

The idea of using specialized hardware for more than just intelligent controllers is not new. The first ideas using specialized hardware centered around using associative memory [2,6,46,49]. Associative processors use an associative memory capable of access by content, not location. This technology (hardware and software of associative memories) has not progressed as quickly as the development of disk technology and VLSI chip technology. Therefore, the idea of associative processors and memory is not being used much in new research.

*1.4.5 Commercial Database Machines.* The concepts for implementing database machines presented to this point all represent hypothetical database machines or machines that have only been prototyped for research. Currently there are only two commercially available database machines [28,59,1], the Britton-Lee and Teradata machines. There have been other companies announce database machines, such as Intel and their iDBP, [57,71] but the machines have either been withdrawn or never marketed.

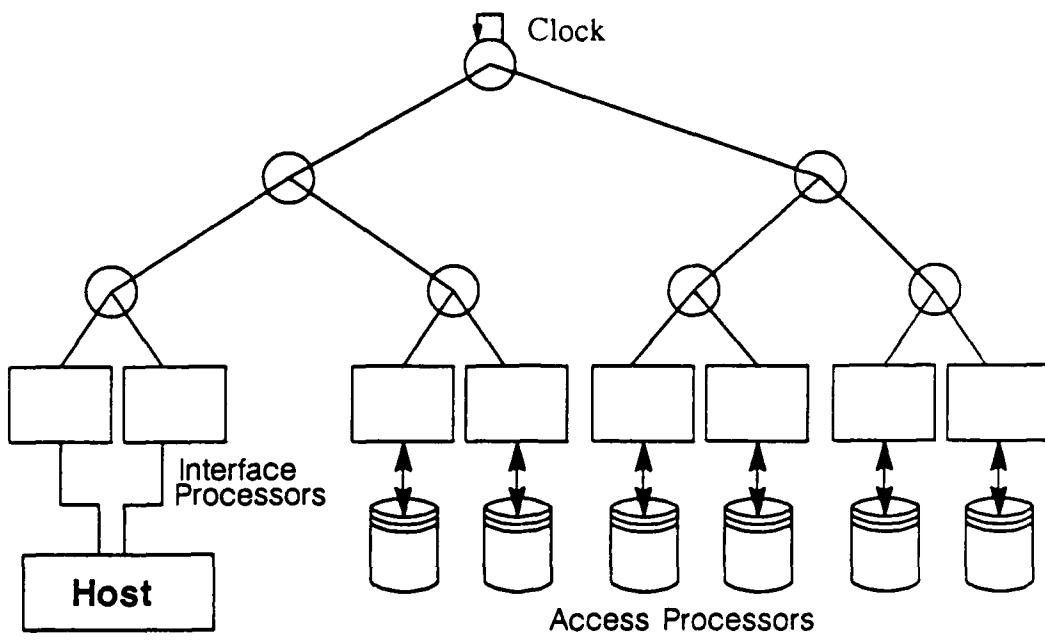


Figure 10. Teradata Ynet Architecture

The Britton-Lee IDM-500 series database machine is the most well known and widely used database machine. The IDM is basically a uniprocessor machine with an optimized disk system. An optional database accelerator is available but it is not known exactly what function the accelerator performs. The Britton-Lee machine has been implemented in several sites and has been tested with seemingly "good" results (good compared with conventional DBMS support) [35,52,84].

The other commercially available database machine is the Teradata DBC/1012. This machine is a multiprocessor system using microprocessors for parallel processing. The microprocessors are connected by Teradata's patented Ynet [1] (see Figure 10). The Ynet is the network that connects the processors plus provides some merging capabilities with some selection logic. This allows the DBC/1012 to use tournament merge operations to implement join operations. Test results of applications using the DBC/1012 are not readily available at this time.

## *1.5 Approach*

Current approaches to the design of database machines are based primarily on intuition and experience. Modeling and simulation of simple queries, to aid in the design process, are becoming more popular. However, the modeling of complex queries/query trees has had very limited exposure, especially modeling query trees for a general type architecture [16]. The current methods provide “good” designs for their purpose, but often fail to fully evaluate the expected performance of an entire system for a full range activity. This lack of a complete evaluation of the performance of the system comes from concentrating on a component or small part of the system rather than designing a complete database machine. Approaching the design in this manner does not provide for evaluation of alternatives outside of the concentration area. Therefore, the design process of a database machine needs to include the modeling of all phases of system activity. The design methodology developed approaches the problem using a method of engineering analysis.

Engineering analysis involves model-building and evaluation. The evaluation of a database machine compares the performance time to complete a query or update of a database. The first step in this analysis is to develop an understanding of the problem by building a mathematical abstraction (i.e., a parametric model) of the important features to be investigated. This model is then used to evaluate alternatives and to show the relationship between components or features of the system. [10]

The design of a database machine has four phases: requirement analysis, the theoretical phase, the analysis phase, and the design. These phases roughly correspond with the engineering design steps [58]. The theoretical/modeling phase was emphasized since previous research only provided limited evaluation and explanation why a particular component or area of a database machine was selected for consideration to improve the design. The theoretical model provides the basis for analysis of various alternatives of components, algorithms, and relationships in a database

system. A model of this type also allows evaluation of the potential of new components or methods after a system has been implemented to determine their feasibility. This type of theoretical model forms the basis of the methodology for designing a database machine and eventually it could be to an expert system to aid in the design of specialized database machines or systems.

The other phases are traditional steps in the design process. The design methodology and tools presented here were developed using these design principles. They also illustrate the use of the theoretical model as a tool to speed the design for a given set of requirements. The following paragraphs provide further explanation of the approach of each phase:

The requirements phase consisted of developing the workload model and system constraints to be used in the detailed design. The workload model was based on two sources of published baseline tests of database machines. The basic workload of a database machine is the retrieval of data in response to user queries for data. Using the published workload models [9.83], there were four basic queries that appeared. These basic query steps were then combined to provide other queries. The four basic query steps are:

1. Select with the results consisting of one or two tuples
2. Select with the results consisting of a larger group of tuples or percentage of the size of the relation
3. Project
4. Joining two relations

These steps are combined to form more complex queries. Although the workload models do not include the other relational operators, the database machine must be capable of handling any query. Therefore, the concentration of the workload model and further phases was on the select, project, and join operators. But implementation for the other operators was provided, to insure that the database machine

provides complete support for any relational query. The workload model also included the need for the database to be maintained. This includes the insert, delete, and modification of tuples as required operations of the database machine.

The next phase after the requirements analysis was the theoretical phase. The purpose of the theoretical phase was to develop a general theoretical model of a database machine. The abstract model was developed incrementally. First, the relational operators were examined to determine the feasibility of relational operators in a parallel processing environment. This also included the relational operators use of different forms of distributed data and the theoretical feasibility of the operators in this environment. Next, abstract model representation for the execution of a single basic query step was developed. The models developed in the theoretical phase are abstract mathematical models to provide the basis for the next phase. The next phase began the analysis of determining the "best" performance model for each of the basic query operator and update operator.

The analysis phase used the abstract models to compare the performance of various operator implementations under various situations. The purpose of this phase was to provide comparison of the performance of operators necessary to execute a user query. The individual operator models were evaluated under various workload conditions to provide a set of parameters under which a "best" model of an operator was determined. These models were then combined with the models for the update operations to determine the best set of implementations for a given user requirement. Then the information gathered during the analysis of the individual operators was combined to form the model of a general multi-step query.

The final phase was the design phase. The design phase took the results of the previous phase and translated them into a system design for a database machine. The system design considers the user attributes of transaction rate, storage requirement, and predictability of access to data [29]. However, the design incorporates flexibility to tailor the system for the individual user requirements by providing modularity of

components and expansion capability. Therefore, the final design phase presented a logical design of a database machine. A final database machine design for implementation would map the logical architecture (determine exact number of processors and disks and their placement in the architecture) for the individual user requirements. This phase would apply the user's storage requirement and query workload using the parameters developed to provide the final query processor design.

### *1.6 Organization*

The sequence of presentation parallels the development of the database machine design process. The design process began by examining the feasibility of applying parallel processing to executing relational operators. Chapter II presents the results. Next, Chapter III, IV, and V present the analytical modeling of the individual operators. Then, the performance models of the update operations are described in Chapter VI. Chapter VII presents the conclusions of the analytical modeling. The effect of considering the execution of multi-step queries and how this changes the analytical model conclusions is presented in Chapter VIII. Finally, Chapter IX presents a logical database machine architecture and describes how to map the logical architecture into a physical design.

## *II. The Feasibility of Relational Operators with Partitioned Relations*

The relational model is based upon mathematical principles that allow proof of correctness of its features [18]. This also provides an ad hoc query capability because the data is not physically related in storage; instead, relationship of data is done by logical connections within the data [17], providing data independence. The following sections use the mathematical principles of the relations to prove the ability of the relational operators to execute with partitioned data.

The retrieval of data from secondary storage continues to be a bottleneck in database machines [22]. The purpose of distributing data is to allow concurrent processing (parallel processing) and retrieval of data by multiple processors and data storage units. This ability to concurrently retrieve and process the data reduces the time to retrieve the data allowing the application needing the data to provide better performance. However, relations of a relational database cannot be arbitrarily distributed without the potential of losing some of the logical connections that are needed to accurately retrieve data. Therefore, the first step is to examine the properties of distributed data.

### *2.1 Data Partitioning*

Relations may be distributed for processing in three ways. The first is storing or processing complete relations (not distributing the data). The second method distributes the relation using horizontal fragments of the relation. The final way distributes the data as vertical fragments. The next section will describe the properties of fragments of relations.

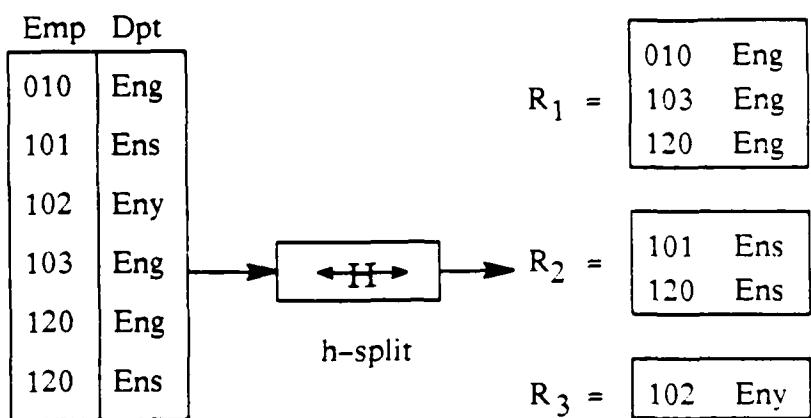


Figure 11. Horizontal Partitioning

## 2.2 Horizontal Partitioning

A relation may be split into fragments either horizontally or vertically. Splitting a relation horizontally results in a subset of the complete set of tuples in each fragment. Thus, horizontally splitting a relation is done with a select type operator, h-split. Figure 11 illustrates horizontal fragmentation with h-split.

The h-split operator produces horizontal fragments of the relation. In Figure 11, the relation was partitioned by department. This produces fragments that consist of tuples of the original relation. The smallest fragment possible by horizontal splitting is a single tuple. Normally, the split is chosen either to provide some logical grouping of the data (i.e., grouping by dept.) or to provide an even distribution of data. In the latter case, the split condition could be a number of tuples to go in each fragment. If the even distribution of data (disjoint sets of tuples) is the goal of the partitioning, each fragment would be  $1/n$  (for  $n$  fragments) of the original relation. To provide the original relation from the fragments, the fragments are combined using the union operator. Therefore, for the set of horizontal fragments,  $R_1, R_2, \dots, R_{n-1}, R_n$ , the complete, original relation is represented as:

$$R = R_1 \cup R_2 \cup R_3 \dots R_{n-1} \cup R_n \quad \text{or} \quad R = \bigcup_{i=1}^n R_i$$

The union operator combines all the horizontal fragments (subsets of tuples) to form the original relation. The normal operation of the h-split provides disjoint fragments. But, the fragments produced do not have to be disjoint. If the fragments are not disjoint sets, the union operator eliminates duplicate tuples to form the proper relation. In the case of the fragments being disjoint sets, the union operator only has to concatenate the fragments to produce the complete relation.

### 2.3 Vertical Partitioning

Vertical fragmentation of a relation splits the relation by attributes. Vertical fragmentation has been widely studied in connection with normalization of relations. Vertical fragmentation without losing information is impossible to guarantee when a relation is normalized to at least third normal form [19]. Therefore, vertical fragmentation in an operational environment seems to have limited practical application. Also, vertical fragmentation does not provide the opportunity for even distribution of data for storage such as the horizontal partitioning. However, the cases of fully indexing a relation have not been fully researched and this closely resembles vertical fragmentation. It is examined here to insure a complete discussion of possible cases and to allow its development for its use in logical applications such as views.

Vertical fragmentation splits the relation by using repeated projections of the original relation. The original relation must be obtainable by joining the fragments without losing data due to eliminating some of the logical connections. Therefore, if the relation has already been normalized, each fragment must contain an attribute(s) that provides a key for the relation to provide valid fragments. Therefore, the smallest vertical fragment possible would be a single attribute that was a key for the relation. Figure 12 shows the vertical fragmentation operator, v-split.

The vertical fragments may be recombined using natural joins to produce the original relation. To recombine several fragments, repeated joins may be necessary.

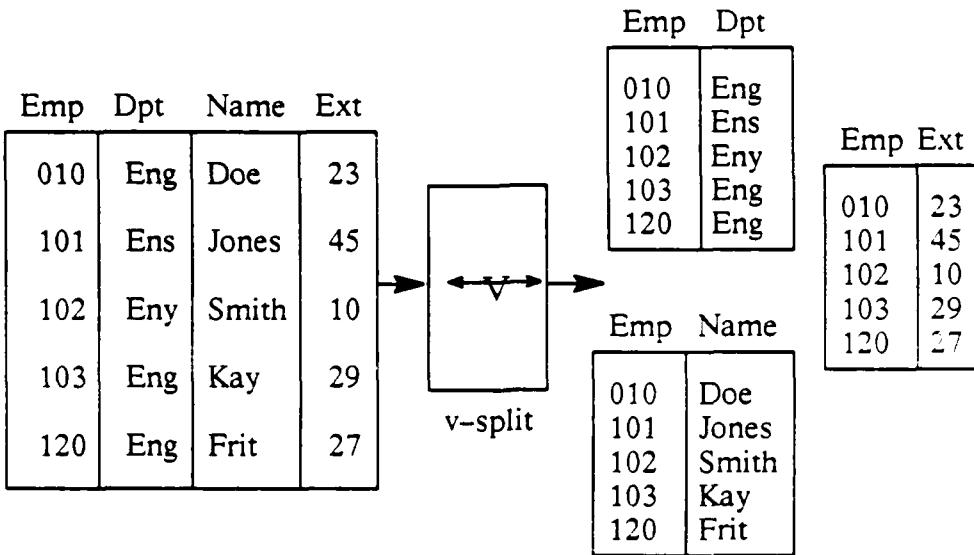


Figure 12. Vertical Partitioning

For the set of vertical fragments,  $R_1, R_2, \dots, R_{n-1}, R_n$ , this is shown by the equation:

$$R = R_1 \bowtie R_2 \bowtie R_3 \dots R_{n-1} \bowtie R_n \quad \text{or} \quad R = \bigotimes_{i=1}^n R_i$$

#### 2.4 Relational Operators

Next, models of the relational operators will explore the processing when the complete relation(s), vertical fragments, or horizontal fragments are provided for processing. First, the mathematical properties of relational operators are examined.

The mathematical properties of relational operators depend upon the equivalence of two expressions. Remembering that a relation is a set of mappings of attributes to values, then a relational algebra expression whose operands are relation variables  $R_1, R_2, \dots, R_k$  defines a mapping from  $k$ -tuples of relations  $(r_1, r_2, \dots, r_k)$ . The mapping results in a single relation which results when each  $r_i$  is substituted for  $R_i$  and the expression evaluated. Two expressions  $E_1$  and  $E_2$  are equivalent if they represent the same mapping. This means when the same relations for identical names are substituted for identical names in equivalent expressions, the results are

the same [79]. Using this definition of equivalence, properties of relational operators can be defined.

Ullman provides the following properties of relational operators [79]. These properties are used by many (i.e., Smith and Chang [73]) for query optimization. Some additional properties will be shown for distributing operators over relation fragments. The laws and properties will be used in the individual discussions of operators to prove their validity for the different types of fragments encountered.

**Property 1** *Joins and Products are Commutative.*

$$E_1 \bowtie E_2 \equiv E_2 \bowtie E_1$$

$$E_1 \times E_2 \equiv E_2 \times E_1$$

**Property 2** *Joins and Products are Associative.*

$$(E_1 \bowtie E_2) \bowtie E_3 \equiv E_1 \bowtie (E_2 \bowtie E_3)$$

$$(E_1 \times E_2) \times E_3 \equiv E_1 \times (E_2 \times E_3)$$

**Property 3** *Cascade of Projections.*

$$\pi_{A_1 \dots A_n}(\pi_{B_1 \dots B_n}(E)) \equiv \pi_{A_1 \dots A_n}(E)$$

where  $A_1 \dots A_n$  contained in  $B_1 \dots B_n$

**Property 4** *Cascade of Selections.*

$$\sigma_{F_1}(\sigma_{F_2}(E)) \equiv \sigma_{F_1} \wedge \sigma_{F_2}(E)$$

Since  $F_1 \wedge F_2 = F_2 \wedge F_1$ , it follows immediately that selections can be commuted.  
i.e.,

$$\sigma_{F_1}(\sigma_{F_2}(E)) \equiv \sigma_{F_2}(\sigma_{F_1}(E))$$

**Property 5 Commuting Selection and Projection.**

If condition  $F$  involves only attributes  $A_1, \dots, A_n$

$$\pi_{A_1, \dots, A_n}(\sigma_F(E)) \equiv \sigma_F(\pi_{A_1, \dots, A_n}(E))$$

More generally, if  $F$  also involves attributes  $B_1, \dots, B_m$  that are not among  $A_1, \dots, A_n$ , then

$$\pi_{A_1, \dots, A_n}(\sigma_F(E)) \equiv \pi_{A_1, \dots, A_n}(\sigma_F(\pi_{A_1, \dots, A_n}, B_1, \dots, B_m(E)))$$

**Property 6 Commuting Selection with Cartesian Product.**

If all attributes mentioned in  $F$  are attributes of  $E_1$ , then

$$\sigma_F(E_1 \times E_2) \equiv \sigma_F(E_1) \times E_2$$

Corollary formed by using rules (1), (4), and (6). When  $F$  is of the form  $F_1 \wedge F_2$ , where  $F_1$  involves only attributes of  $E_1$ , and  $F_2$  involves only attributes of  $E_2$

$$\sigma_F(E_1 \times E_2) \equiv \sigma_{F_1}(E_1) \times \sigma_{F_2}(E_2)$$

Also, if  $F_1$  involves only attributes of  $E_1$ , but  $F_2$  involves attributes of both  $E_1$  and  $E_2$ , we can still assert

$$\sigma_F(E_1 \times E_2) \equiv \sigma_{F_2}(\sigma_{F_1}(E_1 \times E_2))$$

This can also be extended to joins since a join is a product-selection. Therefore, using the attribute involvements of the previous products, we also have:

$$\sigma_F(E_1 \bowtie E_2) \equiv \sigma_F(E_1) \bowtie E_2$$

$$\sigma_F(E_1 \bowtie E_2) \equiv \sigma_{F_1}(E_1) \bowtie \sigma_{F_2}(E_2)$$

$$\sigma_F(E_1 \bowtie E_2) \equiv \sigma_{F_2}(\sigma_{F_1}(E_1 \bowtie E_2))$$

**Property 7** *Commuting selection with a Union.*

Given the expression  $E = E_1 \cup E_2$ , it may be assumed the attributes of  $E_1$  and  $E_2$  have the same names as those of  $E$ , or at least, that there is a given correspondence that associates each attribute of  $E$  with a unique attribute of  $E_1$  and a unique attribute of  $E_2$ . Thus

$$\sigma_F(E_1 \cup E_2) \equiv \sigma_{F_1}(E_1) \cup \sigma_{F_2}(E_2)$$

If the attribute names for  $E_1$  and/or  $E_2$  actually differ from those of  $E$ , then the formulas  $F$  on the right must be modified to use the appropriate names.

**Property 8** *Commuting Selection with a Set Difference.*

$$\sigma_F(E_1 - E_2) \equiv \sigma_{F_1}(E_1) - \sigma_{F_2}(E_2)$$

As in (7), if the attributes names of  $E_1$  and  $E_2$  differ, we must replace the attributes in  $F$  on the right by the corresponding names for  $E_1$ .

**Property 9** *Commuting a Projection with a Cartesian Product.*

$$\pi_{A_1 \dots A_n}(E_1 \times E_2) \equiv \pi_{B_1 \dots B_n}(E_1) \times \pi_{C_1 \dots C_n}(E_2)$$

where  $B_1 \dots B_n$  and  $C_1 \dots C_n$  are contained in  $A_1 \dots A_n$   
and  $B_1 \dots B_n$  and  $C_1 \dots C_n$  are attributes of  $E_1$  and  $E_2$ , respectively.

**Property 10** *Commuting a projection with a Union.*

$$\pi_{A_1 \dots A_n}(E_1 \cup E_2) \equiv \pi_{A_1 \dots A_n}(E_1) \cup \pi_{A_1 \dots A_n}(E_2)$$

**2.4.1 Additional Properties.** The following properties are properties not defined by Ullman [79] because Ullman [79], Smith and Chang [73], and others who described the laws and properties of relational operators were mainly concerned with using them for algebraic manipulations for query optimization. However, these additional properties have been proven in classical set and relation theory [75]. These additional properties are described here because they are used in defining and proving the correctness of the relational operators for partitioned relations. Pelgatti and Schriber [61] were concerned with partitioned data but were not concerned with proving the correctness of all the operators for all partitioned cases.

**Property 11** *Commuting a product with a Union.*

$$A \times (B \cup C) \equiv (A \times B) \cup (A \times C)$$

(See Stanat and McAllister [75] for proof)

Note: It is not true that

$$A \cup (B \times C) \equiv (A \cup B) \times (A \cup C)$$

Since, here it must be assumed that the definition of A must equal the definition of  $B \times C$ , but this does not imply anything about the definitions of B and C being equal to the definition of A. Therefore, this property is not valid for all cases.

**Corollary 12** *Commuting join with union.*

$$A \bowtie (B \cup C) \equiv (A \bowtie B) \cup (A \bowtie C)$$

A join is a product-select-project. Therefore, this property can be easily shown using properties (6), (7), (9), (10), and (11).

Relational operators are the tools used to retrieve data from a relational database. The model of each operator examines its features and capabilities. The relational operators modeled are project, select (restriction), union, difference, Cartesian product, and join. Other relational operators, such as intersection, division, and aggregation, are not examined here because they can be defined in terms of the other relational operators or are not necessary to provide the retrieval operations of a relational complete system [79]. Each operator will also be examined for its processing capability of distributed (fragmented) data. Also, the possibility of combining retrieval operations to make multiple and n-way operations will be explored.

## 2.5 *Select*

The selection operator provides a “horizontal” subset of a given relation. The subset consists of tuples within the given relation that satisfy a given condition(s). The subset of the relation is also a relation because no change is made to the attributes, in particular the key field(s) are not disrupted. Thus, the result still contains only unique tuples that are a subset of the original set of unique tuples that constituted the original relation.

Implementing the select for an unordered relation, the select would scan each tuple of the relation to determine if the tuple satisfies the select condition. The tuples that satisfy the select condition(s) are the results of the select. Figure 13 shows some examples of the select operation. The next sections examine the feasibility of using the select operator with fragments of relations.

**2.5.1 Horizontal Fragment Selects.** Horizontal fragmentation divides the given relation into subsets of tuples, similar to the select. The only difference is that the h-split produces subsets that may be recombined with a union operator to form the original relation. Therefore, the h-split is a specialized case of repeated application of the select operation. This leads to the proof that performing a select upon all

Sample Relation			Select from Sample where A < B and C > 1			Select where A = 'a'		
A	B	C	A	B	C	A	B	C
a	b	1				a	b	1
b	c	3						
c	d	5	b	c	3			
d	a	2	c	d	5			

Figure 13. Examples of the Select Operation

the horizontal fragments of a relation is equivalent to performing the select on the relation. The proof follows as:

$$R = \bigcup_{i=1}^n R_i$$

$$R = R_1 \cup R_2 \cup R_3 \dots \cup R_{n-1} \cup R_n$$

$$\sigma(R) = \sigma(R_1 \cup R_2 \cup R_3 \dots \cup R_{n-1} \cup R_n)$$

$$\sigma(R) = \sigma(R_1) \cup \sigma(R_2) \cup \sigma(R_3) \dots \cup \sigma(R_{n-1}) \cup \sigma(R_n)$$

The proof uses the laws of relational algebra that were presented previously. This allows it to be easily shown that the selection over all the horizontal fragments of a relation is equivalent to performing the selection on the original relation. (For an alternate proof see [62].) The importance of this property is discussed next .

The time to execute the select over a set of horizontal fragments depends upon the number of processors available and the I/O time required to distribute the fragments to processors. The execution of a select by using the h-split operator to form fragments and then executing the select on each fragment is not feasible because the time to execute the h-split would be  $O(n)$  (assuming the relation is size n), which is the same amount of time to process the select. Therefore, to use horizontal fragments it is assumed that the fragments are created as the relation is stored and stored on separate storage devices. Then the time to execute the select

over the fragments is  $n/p$ , where  $p$  is the number of processors available. Thus, if  $p = n$  the time to execute the select would be 1 plus the time required to combine the results.

The results of the selects of the fragments are combined by the union operator to form the final results. The union operator eliminates any duplicates but in the case of the fragments being disjoint sets the union operator simply concatenates the partial results to provide the complete result.

**2.5.2 Vertical Fragment Selects.** Vertical fragmentation causes a unique problem for the selection operator. The problem varies depending upon the conditions expressed for the selection operator. In some cases, the select condition may only need to examine one of the fragments to select the tuples that satisfy the selection condition. But, the results of that selection must then be joined with all the other vertical fragments to produce the complete results. For other selection conditions, fragments may have to be joined before the selection condition may be evaluated (e.g., select where  $A = B$  but  $A$  and  $B$  are contained in separate vertical fragments). Figure 14 shows some examples of selects using vertical fragments.

The properties of relational operators prove the correct results can be produced. This is done as follows:

$$\begin{aligned} R &= \bigtimes_{i=1}^n R_i \\ \sigma_F(R) &= \sigma_F(\bigtimes_{i=1}^n R_i) \\ \sigma_F(R) &= \sigma_F(R_1) \bowtie R_2 \bowtie R_3 \dots \bowtie R_{n-1} \bowtie R_n \\ \sigma_F(R) &= \sigma_{F_1}(R_1) \bowtie \sigma_{F_2}(R_2) \bowtie \sigma_{F_3}(R_3) \dots \bowtie \sigma_{F_{n-1}}(R_{n-1}) \bowtie \sigma_{F_n}(R_n) \end{aligned}$$

Where the attributes of  $F_x$  must be present in the corresponding fragment. If a fragment does not contain an attribute that is part of the expression  $F$ , then  $F_x$  contains nothing so all tuples of fragment  $R_x$  are selected.

The select over vertical fragments does provide the correct results, but it does not seem to meet the goal of fragmentation which is to allow more opportunity for

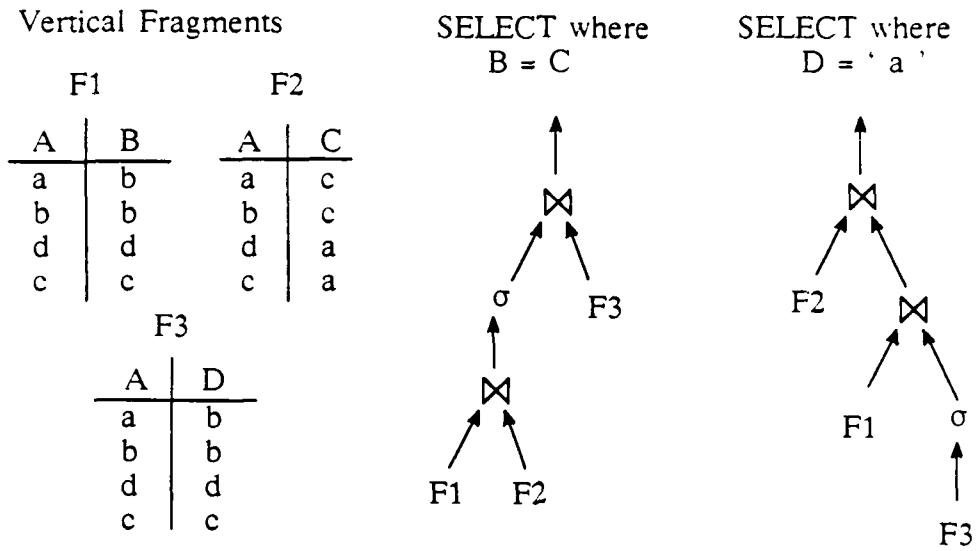


Figure 14. Examples of Selection with Vertical Fragments

parallel processing. This is caused because the select must examine all of one or more fragments which may or may not be less work than doing the select over the entire relation. This coupled with the fact that in a database that was normalized to at least third normal form (normalization is a method of database design and an explanation can be found in many classical database books, such as, Date [19] and Ullman [79]) or more there does not exist much opportunity for vertical partitioning to occur without losing data. Therefore, vertical fragmentation does not provide a good opportunity for optimization of retrievals using the select operator.

## 2.6 Projection

The projection operator extracts attributes from the relation yielding a "vertical" subset of the relation. The projection operator (as defined by Codd [18]) also eliminates duplicate tuples from the results. Duplicate tuples may occur because the projection may eliminate the key attribute(s). Figure 15 shows an example of the projection operation. In some instances, the duplicate elimination may be delayed until after other operations (i.e., select, join, even another project). This does not

Sample Relation			Project from Sample [B,C]	
A	B	C	B	C
a	b	1	b	1
b	c	3	c	3
c	b	1	a	2
d	a	2		

\*Notice that the key was removed by the projection, so duplicates were removed

Project from Sample [A,C]		
A	C	
a	1	*This projection did not affect
b	3	the key, so no
c	1	duplicates were
d	2	introduced

Figure 15. Examples of the Projection Operation

cause any variation in the final results other than allowing more data to be handled than necessary.

The implementation of the projection operator scans the relation retaining only the attributes indicated in the command. For a relation of size  $n$ , the time required to execute a projection is  $n$  plus the time required to eliminate duplicates. This assumes using a single processor. The elimination of duplicates may add  $O(m \log m)$ , where  $m$  is the size of the results. This is derived from the normal method of removing the duplicates which is sorting. Other methods of duplicate removal might compare each resultant tuple with each other resultant tuple but this would be  $O(m*m)$ , which is more time consuming.

*2.6.1 Horizontal Fragmentation Project.* The projection on horizontal fragments of a relation provides the opportunity for greater parallel processing. Like the select operator, the time to perform the project would be  $n/p$ , where  $p$  is the number of processors available. Duplicate removal is an integral part of the union operator that must be used to recombine the fragments to provide the complete relation; thus, eliminating the need for duplicate removal at each node. It is shown

below that projection over horizontal fragments is valid.

$$R = \bigcup_{i=1}^n R_i$$
$$R = R_1 \cup R_2 \cup R_3 \dots \cup R_{n-1} \cup R_n$$
$$\pi(R) = \pi(R_1 \cup R_2 \cup R_3 \dots \cup R_{n-1} \cup R_n)$$
$$\pi(R) = \pi(R_1) \cup \pi(R_2) \cup \pi(R_3) \dots \cup \pi(R_{n-1}) \cup \pi(R_n)$$

**2.6.2 Vertical Fragmentation Project.** Vertical fragmentation distributes the relation by projecting out attributes for each fragment. Further application of the projection operator may cause loss of data because the keys used to recombine the fragments may be eliminated. Therefore, the fragments may have to be combined before the projection is performed. If the attributes eliminated were not needed as keys for the recombination of the fragments, the project was successful. Therefore, vertical fragmentation could immediately provide the needed results or the vertical fragments could provide the correct results without reforming the entire relation and then performing the projection. Thus, the projection with vertical fragments does guarantee correct results without special consideration which may require providing the complete original relation before the projection can occur. Therefore, vertical fragmentation can not be used for possible performance improvement of the projection operation without special considerations.

## 2.7 Cartesian Product

The Cartesian Product (product) constructs a relation by concatenating tuples from two relations (written as,  $R \times S$ , for relations  $R$  and  $S$ ). The product concatenates each tuple from one input relation with each tuple from the other input relation. This is shown in Figure 16. Normally, a product is not used; instead, a join operation is performed. The join operation is defined to be a product combined with a select and a project, that yields the desired limited results. The term join may be confusing because it implies several different situations of limiting results. The first, the natural join is a product, a select (performing a select on common

Relation R		Relation S			R × S				
A	B	C	D	E	A	B	C	D	E
a	b	a	b	c	a	b	a	b	c
c	d	d	e	f	a	b	d	e	f
		g	h	i	a	b	g	h	i
					c	d	a	b	c
					c	d	d	e	f
					c	d	g	h	i

Figure 16. The Product Operation

attributes), and a project to remove any duplicate attributes. The natural join is the join used in the vertical fragmentation. The term join is also used to define when select is performed on to find equal attribute values on attributes defined on a common domain but are not the same attribute name. This situation is called an equi-join. And the last join situation is when the select criteria compares for other than an equality condition. The equi-join of the join will be examined in a later section. However, it must be remembered that the join used to combine horizontal fragments is the natural join.

The product requires each tuple of the relations to be combined. This is an  $O(m*n)$  operation, where  $m$  and  $n$  are the number of tuples in the relations to be operated on. However, the performance consideration of the operation may be extended in many cases to consider data access time. The goal of parallel processing is to reduce the time required to execute the product from  $O(m*n)$  to  $O((m*n)/p)$ , where  $p$  is the number of processors employed.

**2.7.1 Horizontal Fragment Product.** The following shows that using horizontal partitions for the Cartesian Product is deterministic:

$$\begin{aligned}
 R &= \bigcup_{i=1}^n R_i \quad \text{and} \quad S = \bigcup_{i=1}^m S_i \\
 R \times S &= (R_1 \cup R_2 \cup \dots \cup R_{n-1} \cup R_n) \times S \\
 &= (R_1 \times S) \cup \dots \cup (R_{n-1} \times S) \cup (R_n \times S)
 \end{aligned}$$

$$\begin{aligned}
&= (R_1 \times (S_1 \cup \dots \cup S_{m-1} \cup S_m)) \\
&\quad : \\
&\quad \cup (R_{n-1} \times (S_1 \cup \dots \cup S_{m-1} \cup S_m)) \\
&\quad \cup (R_n \times (S_1 \cup \dots \cup S_{m-1} \cup S_m)) \\
&= (R_1 \times S_1) \cup \dots \cup (R_1 \times S_{m-1}) \cup (R_1 \times S_m) \\
&\quad : \\
&\quad \cup (R_{n-1} \times S_1) \cup \dots \cup (R_{n-1} \times S_{m-1}) \cup (R_{n-1} \times S_m) \\
&\quad \cup (R_n \times S_1) \cup \dots \cup (R_n \times S_{m-1}) \cup (R_n \times S_m)
\end{aligned}$$

The problem with doing the product with horizontal fragments is the number of individual products necessary to complete the operation. This may over complicate the problem because of the added control necessary to read and move all of the fragments to the correct place for processing. However, the performance is reduced from  $m*n$  to  $(m/p)*n$  by using  $p$  processors. If the problem in this completely distributed form becomes too complex, one of the intermediate steps in the proof may be used to complete the problem. Using one complete relation and fragments of the other relation is an example of this. This may provide a more appropriate distribution of the problem.

**2.7.2 Vertical Fragment Product.** The product with vertical fragments is deterministic because the product does not eliminate any data from either relation. This leaves the keys for each fragment and relation in place allowing the recombination of fragments. The proof of this is shown below. (The proof uses property of the join that it is defined as a product-select-project or in mathematical notation  $R \bowtie S = \pi(\sigma(R \times S))$ .)

$$\begin{aligned}
R &= \bigotimes_{i=1}^n R_i \quad \text{and} \quad S = \bigotimes_{i=1}^m S_i \\
R \times S &= (R_1 \bowtie \dots \bowtie R_{n-1} \bowtie R_n) \times (S_1 \bowtie \dots \bowtie S_{m-1} \bowtie S_m)
\end{aligned}$$

$$\begin{aligned}
&= \pi_R(\sigma_R(R_1 \times \cdots R_{n-1} \times R_n)) \times \pi_S(\sigma_S(S_1 \times \cdots S_{m-1} \times S_m)) \\
&= \pi_{RS}(\sigma_{RS}((R_1 \times \cdots R_{n-1} \times R_n) \times (S_1 \times \cdots S_{m-1} \times S_m))) \\
&= \pi_{RS}(\sigma_{RS}(R_1 \times \cdots R_{n-1} \times R_n \times S_1 \times \cdots S_{m-1} \times S_m))
\end{aligned}$$

The grouping of operations could be done in any fashion now since the product is associative and commutative. If  $m = n$  one grouping would be:

$$\pi_{RS}(\sigma_{RS}((R_1 \times S_1) \times \cdots (R_{n-1} \times S_{m-1}) \times (R_n \times S_m)))$$

Then moving the individual select and project conditions to the grouping they operate on provides the general form:

$$(R_1 \times S_1) \bowtie \cdots (R_{n-1} \times S_{m-1}) \bowtie (R_n \times S_m)$$

It should be noted that some of the natural joins, in the general form, do not have attributes to compare and the join only performs the product portion of the join. Therefore, the grouping of operations may affect the efficiency of the operations by reducing the size of the results at intermediate steps, thus utilizing the reduction of the select and project of the natural join wherever possible.

The performance time of the product with vertical fragments is similar to the time for a product with horizontal fragments except that the vertical fragments require that the fragments be combined with a join operation which is a modification of the product. Therefore, the product of vertical fragments will not always perform better than doing the product with relations stored in a single entity.

## 2.8 Join

The join operator is a Cartesian Product combined with a select and in some situations a project. The join normally selects only the tuples from the product where two attributes (one attribute from each input relation) have an equal value. This type of join is called an equi-join but in most of the current literature this is the type

Relation R		Relation S			JOIN R and S where A = C or R $\bowtie$ S				
A	B	C	D	E	A	B	C	D	E
a	b	a	b	c					
d	c	d	e	f					
		g	h	i					

Figure 17. The Join (Equi-Join) Operation

of join referred to by the join operation. The natural join used to combine horizontal fragments is a special situation of the equi-join where the attributes compared have the same attribute names and duplicate attributes are removed by projection in the results. The join may also indicate unequal conditions for the selection but these cases are normally ignored because the most efficient implementations of unequal joins are the same as the implementation of the product operator. Therefore, it is assumed that join means equi-join for this discussion but not necessarily natural join (See Figure 17).

The implementation of the join has been widely researched because it is the most time consuming operation that is used extensively in queries. The various implementations are done for many different architectures and data storage schemes. These will be examined later. The purpose now is to provide proof that the join can operate correctly where the data is fragmented.

*2.8.1 Horizontal Fragment Join.* The joining of two relations, where the relations have been partitioned horizontally, is possible but as the proof shows, the number of fragment joins becomes very large. The large number of joins involving the fragments may cause a large overhead to control the joins. This has caused some [23] to use the intermediate form, of considering fragments from one relation but treating the other relation as a single entity. The following shows that joining

vertical fragments is feasible.

$$\begin{aligned}
 R &= \bigcup_{i=1}^n R_i \quad \text{and} \quad S = \bigcup_{i=1}^m S_i \\
 R \bowtie S &= (R_1 \cup R_2 \cup \dots \cup R_{n-1} \cup R_n) \bowtie S \\
 &= (R_1 \bowtie S) \cup \dots \cup (R_{n-1} \bowtie S) \cup (R_n \bowtie S) \\
 &= (R_1 \bowtie (S_1 \cup \dots \cup S_{m-1} \cup S_m)) \\
 &\quad \vdots \\
 &\quad \cup (R_{n-1} \bowtie (S_1 \cup \dots \cup S_{m-1} \cup S_m)) \\
 &\quad \cup (R_n \bowtie (S_1 \cup \dots \cup S_{m-1} \cup S_m)) \\
 &= (R_1 \bowtie S_1) \cup \dots \cup (R_1 \bowtie S_{m-1}) \cup (R_1 \bowtie S_m) \\
 &\quad \vdots \\
 &\quad \cup (R_{n-1} \bowtie S_1) \cup \dots \cup (R_{n-1} \bowtie S_{m-1}) \cup (R_{n-1} \bowtie S_m) \\
 &\quad \cup (R_n \bowtie S_1) \cup \dots \cup (R_n \bowtie S_{m-1}) \cup (R_n \bowtie S_m)
 \end{aligned}$$

The performance improvement possible using horizontal fragments with the join operation is difficult to define. However, given  $p$  processors, each processor would compare  $(x/p) * y$  blocks versus the  $x * y$  comparisons required of a single processor. The results would have to then be combined by performing the union operation. If the special consideration of having disjoint partitions is considered, the union now becomes only a concatenation which reduces the time required to perform the union. Therefore, the horizontal fragmentation combined with the join operation may utilize multiple processors to reduce the execution time.

*2.8.2 Vertical Fragment Join.* Joining vertical fragments of two relations involves multiple levels of joins. The first level join would join the fragments of the different relations. The higher level joins would join the results of the low level joins to produce the complete results. This requires that the order of join vertical fragments be maintained so the proper join attributes are available. Also, the low level

joins may not be possible because the fragments of the two relations may not share the join attribute. This makes the joining of vertically partitioned relations a very demanding operation to insure correct order and positioning the fragments so the fragments have the correct join attribute. The following shows the derivation of the mathematical scheme for joining vertical fragmented relations.

$$\begin{aligned} R &= \bigotimes_{i=1}^n R_i \quad \text{and} \quad S = \bigotimes_{i=1}^m S_i \\ R \bowtie S &= (R_1 \bowtie \cdots \bowtie R_{n-1} \bowtie R_n) \bowtie (S_1 \bowtie \cdots \bowtie S_{m-1} \bowtie S_m) \\ &= (\sigma_{RS}(R_1 \bowtie \cdots \bowtie R_{n-1} \bowtie R_n \bowtie S_1 \bowtie \cdots \bowtie S_{m-1} \bowtie S_m)) \end{aligned}$$

The grouping of operations could be done in any fashion now since the product is associative and commutative. If  $m = n$  one grouping would be:

$$(R_1 \bowtie S_1) \bowtie \cdots \bowtie (R_{n-1} \bowtie S_{m-1}) \bowtie (R_n \bowtie S_m)$$

Performance improvement through the use of multiprocessing for the join with vertical fragments is very difficult to define due to the various groupings of processing that could be used to complete the operation. Multiple processors could be used to compare the individual fragments but then the fragment join results must be joined. This combined with the constraint of some of the joins being products makes the performance improvement through the use of multiple processors difficult to compute for the general case.

## 2.9 Union

The union operator combines the tuples of two relations, eliminating any duplicate tuples. The only requirement of the union operator is that the relations are union compatible. Union compatibility means the relations have an equal definition, which requires the relations to have the same number of attributes and the same domains for corresponding attributes. Figure 18 illustrates the union of two relations.

Relation A			Relation B			A $\cup$ B		
R	S	T	R	S	T	R	S	T
a	b	c	a	d	c	a	b	c
b	c	d	b	c	d	a	d	c
e	f	g	c	a	b	b	c	d
			h	i	j	c	a	b
						e	f	g
						h	i	j

Figure 18. The Union Operation

*2.9.1 Horizontal Fragment Union.* The union of relations that have been partitioned into horizontal fragments involves the union of unions. Since the union operator is both commutative and associative, there exists no constraints to the order in which the unions are performed. Therefore, the union of horizontal fragments is unconstrained and feasible.

*2.9.2 Vertical Fragment Union.* The union of vertically fragmented relations involves the union of joined fragments. To distribute this for parallel processing would involve distributing the union over join. This coupled with the fact that the fragments to be joined do not have to have the same definition causes the distributing of the union over the join to be infeasible.

$$\begin{aligned}
 R &= \bigwedge_{i=1}^n R_i \quad \text{and} \quad S = \bigwedge_{i=1}^m S_i \\
 R \cup S &= (R_1 \bowtie R_2 \bowtie \cdots R_{n-1} \bowtie R_n) \cup S \\
 &\neq (R_1 \cup S) \bowtie \cdots (R_{n-1} \cup S) \bowtie (R_n \cup S)
 \end{aligned}$$

This is not equal because the union requires the same attributes for each input relation (or fragment). Therefore, only R and S are sure to have the same attribute definitions and unions of the fragments are not always possible. In the case where the relations were partitioned exactly the same, unions of the corresponding fragments would be possible.

Relation A			Relation B			(A - B)			(B - A)		
R	S	T	R	S	T	R	S	T	R	S	T
a	b	c	a	d	c	a	b	c	a	d	c
b	c	d	b	c	d	e	f	g	c	a	b
e	f	g	c	a	b				x	y	z
h	i	j	h	i	j						
			x	y	z						

Figure 19. Difference operation

### 2.10 Difference

The difference operator provides all the tuples of the first relation that are not duplicated in the second input relation. This requires the input relations to be defined with the same attributes (union compatible). Since the results are the tuples of the first relation that did not match a tuple in the second relation, difference is not commutative. Figure 19 illustrates the action of the difference operator.

**2.10.1 Horizontal Fragment Difference.** The key to the difference operator is that it provides all the tuples from the first relation that do not match with a tuple of the second relation. Because of this, it may be possible to perform a difference with horizontal fragments. The constraints to successfully completing the difference with horizontal fragments are the relations must be union compatible and each fragment of the first relation must be operated on by the entire second relation. This provides results that may be combined to provide correct complete results. The following shows this and explains possible further extensions to this.

$$\begin{aligned}
 R &= \bigcup_{i=1}^n R_i \quad \text{and} \quad S = \bigcup_{i=1}^m S_i \\
 R - S &= (R_1 \cup R_2 \cup \dots \cup R_{n-1} \cup R_n) - S \\
 &= (R_1 - S) \cup \dots \cup (R_{n-1} - S) \cup (R_n - S)
 \end{aligned}$$

Note: The results in each case have been determined by comparing a fragment with an entire relation. Since, the second relation is the controlling factor this provides accurate results. Figure 20 shows an example of this using the data from the example in Figure 19. The second relation can also be fragmented and still provide a correct response without first recombining the fragments to form the whole relation. The following shows how this is possible.

The equation, using the fragmented S is:

$$\begin{aligned}
 R - S &= (R_1 - (S_1 \cup \dots \cup S_{m-1} \cup S_m)) \\
 &\quad : \\
 &\quad \cup (R_{n-1} - (S_1 \cup \dots \cup S_{m-1} \cup S_m)) \\
 &\quad \cup (R_n - (S_1 \cup \dots \cup S_{m-1} \cup S_m))
 \end{aligned}$$

Looking at only the line of the equation using  $R_1$ , you might expect:

$$(R_1 - S_1) \cup \dots \cup (R_1 - S_{m-1}) \cup (R_1 - S_m)$$

but this does not work because combining the results with the union operator reintroduces tuples that had matched and been eliminated in one of the fragment differences. In this case, the tuples desired are the ones that are produced in every result, with the fragment of the first relation being used as the control element. Applying the theorem of set theory [75],  $A - (B \cup C) = (A - B) \cap (A - C)$ , the correct operation is the intersection of the results of the fragment differences for a given fragment of the first relation. Then the results of these differences are combined with a union to provided the final result (Figure 21 shows an example). The final equation is in the form:

$$\begin{aligned}
 R - S &= ((R_1 - S_1) \cap \dots \cap (R_1 - S_{m-1}) \cap (R_1 - S_m)) \\
 &\quad :
 \end{aligned}$$

For relation A and B, with A fragmented:

Relation A <sub>1</sub>			Relation A <sub>2</sub>			Relation A <sub>3</sub>			Relation B		
R	S	T	R	S	T	R	S	T	R	S	T
a	b	c	b	c	d	e	f	g	a	d	c
						h	i	j	b	c	d
									c	a	b
									h	i	j
									x	y	z

From above:

$$A - B = (A_1 - B) \cup (A_2 - B) \cup (A_3 - B)$$

A <sub>1</sub> - B			A <sub>2</sub> - B			A <sub>3</sub> - B		
R	S	T	R	S	T	R	S	T
a	b	c				e	f	g

or

$\emptyset$

then combining the results with the union operation

A - B =			R	S	T
			-	-	-
a	b	c			
e	f	g			

Figure 20. Difference with Horizontal Fragments and a Relation

$$\cup ((R_{n-1} - S_1) \cap \dots (R_{n-1} - S_{m-1}) \cap (R_{n-1} - S_m))$$

$$\cup ((R_n - S_1) \cap \dots (R_n - S_{m-1}) \cap (R_n - S_m))$$

**2.10.2 Vertical Fragment Difference.** The difference of vertical fragments is not feasible except for one special case. It is not feasible because the fragments are not necessarily union compatible. For the difference of vertical fragments to operate, the relations must be partitioned to provide compatible fragments. This means that the definition of the corresponding fragments must be union compatible (i.e.,  $R_1 = S_1, R_2 = S_2, \dots$ ). In this case, the difference may be accomplished using vertical fragments but only for this limited case.

$$\begin{aligned} R &= \bigwedge_{i=1}^n R_i \quad \text{and} \quad S = \bigwedge_{i=1}^m S_i \\ R - S &= (R_1 \bowtie R_2 \bowtie \dots \bowtie R_{n-1} \bowtie R_n) - S \\ &\neq (R_1 - S) \bowtie \dots \bowtie (R_{n-1} - S) \bowtie (R_n - S) \end{aligned}$$

This is not equal because the fragments are not union compatible with the entire relation.

## 2.11 Multi-way Operations

The purpose of multi-way or n-way operations is to extend binary operations to handle more than two inputs. Since they are defined as binary operations, this literally cannot happen; however, by using partial results from one step as an input to the next step, it is considered a multi-way operation. An example of this type operation is the addition problem  $12 + 23 + 34$ . When you add these numbers, you might add in this pattern,  $(2 + 3 + 4) + (10 + 20 + 30)$ . This provides the same result but allows you to add only single digit numbers since we logically maintain the tens place. This multi-way operation is more efficient for us to handle than doing  $(12 + 23) + 34$ . Such is the purpose of determining if any of the binary relational operators can be done as a multi-way operator.

Difference using horizontal fragments

$$A - B = \left( (A_1 - B_1) \cap (A_1 - B_2) \cap (A_1 - B_3) \right) \\ \cup \left( (A_2 - B_1) \cap (A_2 - B_2) \cap (A_2 - B_3) \right) \\ \cup \left( (A_3 - B_1) \cap (A_3 - B_2) \cap (A_3 - B_3) \right)$$

with fragments

$A_1$	$A_3$	$B_1$	$B_2$
R   S   T	R   S   T	R   S   T	R   S   T
a   b   c	e   f   g	a   d   c	c   a   b
h   i   j	b   c   d	b   c   d	h   i   j
$A_2$		$B_3$	
R   S   T		R   S   T	
b   c   d		x   y   z	

$$A_1 - B_1 \cap A_1 - B_2 \cap A_1 - B_3 = A_1 - B$$

R   S   T	R   S   T	R   S   T	R   S   T
a   b   c	a   b   c	a   b   c	a   b   c

$\cup$

$$A_2 - B_1 \cap A_2 - B_2 \cap A_2 - B_3 = A_2 - B$$

$\emptyset$	R   S   T	R   S   T	$\emptyset$
	b   c   d	b   c   d	

$\cup$

$$A_3 - B_1 \cap A_3 - B_2 \cap A_3 - B_3 = A_3 - B$$

R   S   T	R   S   T	R   S   T	R   S   T
e   f   g	e   f   g	e   f   g	e   f   g
h   i   j		h   i   j	

Therefore, after the union

$$A - B = \begin{array}{|c|c|c|} \hline R & S & T \\ \hline a & b & c \\ \hline e & f & g \\ \hline \end{array}$$

Figure 21. Difference with Horizontal Fragments

The first properties that a binary operator must have before being considered in a multi-way environment are the commutative and associative properties. These properties allow the order of processing to not affect the results. Of the binary relational operators: product, join, union, and difference, difference is not commutative. This eliminates the difference operator from consideration. The other operators are commutative and associative.

The associative and commutative properties provide the opportunity that an operation may be feasible for multi-way situations but does not guarantee it. First, it must be remembered that join, union, and product require one relation to be compared or combined with another relation (this actually means each tuple must be compared or combined). The union compares complete tuples and all of the tuples must have the same definition. Therefore, there is no constraint for the union process, making it feasible for multi-way operation. The product also has no restriction on the definition of the relations it operates on, making it feasible. The join does need the restriction that the inputs share a join attribute. If the inputs do not contain the join attribute, the join must default to a product and later joins must join over all the common attributes, to allow the join operation to be feasible. The actual implementation of a multi-way operation is done by the order of operation. A simple example is the union of three relations. One method of performing the union is to sort the inputs and then compare the top value of each ordered relation to see if they are duplicates. To extend this to a multi-way operation, the comparison would be of  $x$  values where  $x$  is the number of input relations. The process is not an actual multi-way operation because only two values can be compared at a time but like the mathematical example presented earlier a method of grouping the processing. This means that even though the union, join, and product may be feasible for multi-way operations, the efficiency of the actual implementation of the operation may be better suited to a series of binary operations rather than doing a multi-way operation.

	Horizontal Fragments	Vertical Fragments
Select	Yes	Yes
Project	Yes	No*
Join	Yes	Yes
Product	Yes	Yes
Union	Yes	No*
Difference	No*	No*

\* may be feasible for special cases or under certain constrained conditions

Table 1. Feasibility of Operators with Fragments

### 2.12 Conclusions

The discussions of the retrieval operators proved the feasibility of using fragments with the operators. Table 1 summarizes the feasibility of a given relational operator with fragmented data.

The purpose of exploring fragmentation was to provide the basis for distributing the storage and processing of the relations of a relational database. This is important because the time required to retrieve data from secondary storage is the major bottleneck of a database system [22]. By partitioning the data to several secondary storage devices (disk), each device needs only to retrieve a portion of the data, assuming the data is evenly distributed, providing the opportunity for faster database operations. If the data has to be recombined into the original relation before processing, the data would have to be stored back on disk, eliminating the improvement in performance caused by the distributed data storage. Therefore, it is important that the fragments may be processed independently to maintain the speedup gained from the distributed data.

Cases such as the selection with horizontal fragments illustrate the potential improvements possible by using fragmented relations and multiple storage devices and multiple processors. The first step of the select is to get the tuples from the disk for processing. If there are n disks, each with an equal portion of the data, the

data can be retrieved in  $1/n$  the time of a single disk. And if there is a processor associated with each disk, it can process the data as fast as it is retrieved. However, for the case of the selection with the vertical fragments, only one fragment contains the condition to be compared for the select. This causes a smaller amount of data to be retrieved but any processing gain is reduced and depends upon the specific case.

The potential for improved retrievals by parallel processing depends upon two factors: distributing the data for storage and the capability to process the fragments independently. When the operation is not feasible with fragments, such as the difference operation, the fragments must be recombined before processing. This reduces the efficiency of using distributed data. Table 1 shows that the difference operator must operate on the original relations to insure the correct response. But, all the other operations are feasible with horizontal fragments. However, vertical fragments cause difficulty in processing with several operators. This causes the horizontal partitioning of fragments to better suited for exploring potential performance improvements through the use of parallel processing. Therefore, this is the starting point for further research into using distributed storage and multiprocessing to design a database machine for improved database operations.

The results presented indicate that horizontal partitioning provides more opportunity for improvements in data retrieval. The vertical partitioning of relations constrains the relational operations by the lack of union compatibility for some cases and the constant concern to retain the key so the logical connections for data retrievals are not lost. Therefore, only horizontal partitioning is considered in the further evaluation.

### *III. Modeling the Performance of the Select Operator*

A database system is very seldom measured by ease of use or lack of or abundance of features. Instead, the primary method of comparison is performance time. Performance time consists of the time necessary to compile a user query, retrieve the necessary data using relational operators, and either store the results for later user use or send the data to the user's terminal. Obviously, the major factor in the performance of a database system is the retrieval operation. Therefore, this chapter explores the many components of retrieval operations and specifically the select operator. Later chapters discuss the project and join operators and updating the data in the database.

The retrieval operations consists of reading the data from secondary storage and evaluating the data to determine if it satisfies the retrieval criteria (the query conditions). This over-simplifies the retrieval process. There are several factors of performance time of retrievals: the operator necessary for the retrieval (i.e., select, project, join), the structure of the data in secondary storage, the number of secondary storage units, the number of processors units and the physical capabilities of each processor, the volume of data that must be retrieved and evaluated, the algorithms used to control the retrievals, and the number of users trying to use the system. Obviously, it is impossible to examine all possible combinations of the performance factors. Therefore, to allow the evaluation of various factors and their interactions some simplification must be done. The first step taken is to evaluate performance in a dedicated environment. This means that the system is processing a query consisting of only one relational operator for only one user.

The relational operators used for the retrievals can be select, project, join, union, difference, product, division, or intersection. However, the majority of the retrievals can be completed using only select, project, and join. Therefore, these are the relational operators that will be used for the evaluation of the performance

effects of other features. Since performance is a measure of time, the exploration of performance will consist of the development of equations expressed in parameters of hardware performance measures. This will allow improved hardware parameters to be substituted to evaluate the effect of new and improved hardware. The discussion will be divided into four sections. These sections will evaluate the operator for the four different hardware configurations possible. These configurations are: single processor-single disk (disk here means secondary storage since at this time disk storage is the only viable method of secondary storage), single processor-multiple disks, multiple processors-single disk, and multiple processors-multiple disks.

All of the architectures are assumed to have no restrictions. This means that for the multiple processors-single disk environment that all of the processors can access the disk and that each processor could communicate with each other processor (communication may not be direct but can be accomplished through intermediate processors). The processor communication is also assumed for the multiple processors-multiple disks architecture. The multiple disks of this environment are assumed to support all processors. Thus, no time delay for accessing the disk is included in the multiple processors-multiple disks models that follow.

The two final features that must be evaluated within each section are the data storage structure and the algorithm used for the relational operator. The purpose of this chapter is to determine the effects and performance of various algorithms with different storage structures. First, the different potential data structures will be examined.

### 3.1 Data Storage Structures

Data structures for a relational database system are based upon the relation. A relation is defined as follows:

**Definition** Given a collection of sets  $D_1, D_2, \dots, D_r$  (not necessarily distinct),  $R$  is a *relation* on those  $n$  sets if it is a set of ordered  $n$ -tuples  $\langle d_1, d_2, \dots, d_n \rangle$ .

such that  $d_1$  belongs to  $D_1$ ,  $d_2$  belongs to  $D_2, \dots, d_n$  belongs to  $D_n$ . Sets  $D_1, D_2, \dots, D_n$  are the *domains* of R. An *attribute* represents the use of a domain within a relation.

The previous chapter showed how relations may be partitioned for processing and it was concluded that vertical partitioning of data did not provide any advantages for implementing parallel processing. Therefore, it is assumed that the only partitioning of data done will be horizontal partitioning. Next, the different structures for a single data store will be discussed and in the following section the structures of multiple data stores will be examined.

*3.1.1 Single Data Storage Structures.* The single disk or data store does not allow any potential for partitioning of data to allow parallel retrieval of the relation. Thus, data may be stored in an arbitrary manner which will be called unordered or in a sequential order (sorted order) which will be called ordered. However, one additional feature may be present. That is an index of the data. It is assumed that any index involved here can provide a logarithmic type tree structure for the index and that the leaves of the index are the only place that the actual index is contained. The logarithmic type tree models any type of index tree structure by changing the parameter that depicts the number of keys contained in each block of the index. By only allowing the leaves to hold the tuple addresses, the leaves can be accessed to provide sequential accessing of the key values. Therefore, the combinations to be examined for single disk cases are: 1) unordered, no index; 2) ordered, no index; 3) unordered, indexed; and 4) ordered and indexed.

Indexed means that an index exists for the attribute(s) necessary for the relational operator. If an index exists but not for the proper attribute(s) for the operator, the corresponding unindexed case provides the appropriate performance time evaluation. It is assumed that the index is stored on disk. Therefore, all the possible conditions are covered by the four cases.

**3.1.2 Multiple Data Storage Structures.** Multiple disks provide many opportunities for distributing the data. The first cases are obviously the cases where the relations are not partitioned and are stored on a single disk unit as described above. It is assumed that only horizontal partitioning (see previous chapter) is used for distributing the relation among data stores. The purpose of distributing the data among data stores is to reduce the performance time by allowing parallel retrievals of the data. Therefore, any method of dividing the data must provide approximately even distribution across the disks. If the method of partitioning the data does not provide an even distribution, then the retrieval parameters are the same as the single disk case, eliminating the advantage of the parallel retrieval of the multiple disks.

There are three primary methods of partitioning a relation across the data stores. The first method provides an even distribution by using a round robin assignment method. This insures an even distribution but does not provide any grouping of the data, so this method is called the unordered distribution. The method called the ordered distribution is accomplished by maintaining the relation in sorted order with the number of blocks evenly distributed among the disks. Maintaining the even distribution for new insertions may cause a reorganization of the relation requiring the reading and writing of several portions of the relation. Potentially, for the worst case condition, the entire relation must be read and then written back to disk to insert a single tuple.

Another method of distributing the data is by hashing each tuple and assigning certain boundaries to each disk. This makes each disk similar to a bucket in a bucket sort. This method does not require reorganization like the ordered distribution did but it does provide some grouping of the data by value, unlike the round robin distribution. However, the bucket distribution does not guarantee an equal distribution if the boundaries of the buckets are fixed values. If an even distribution is necessary, the bucket boundaries will have to be readjusted, requiring a reorganization of the data. The bucket or hashing method also allows the individual buckets to

be maintained in an ordered or sorted manner if this will provide any performance advantage.

The final storage structure that can be added for multiple data stores is some form of index. Obviously, each disk may have its own index. But, there may be centralized indices for all of the disks. If an index is centralized, then we have the same case as the single data store with index until the actual retrieval of the tuples occurs. This and other considerations will be examined during the development of each performance equation.

### 5.2 Select Performance Models

The select operator restricts the results to only the tuples that satisfy the selection condition. For this evaluation two different cases will be considered. The first, called MT, where MT stands for "many tuples" in the results. This selection case results when the selection criteria desires a range of values or the given condition applies to many tuples. A simple example of this would be a query that desired all the names of employees of a given department when the department had 1000 employees.

The other case is the FT case. The FT case is when the results contain a "few tuples" (such as, 10 or fewer tuples and always less than one block of results). The implication of this type of select case is that a very specific piece of information is desired. The FT selection is a special case of the MT case. It is included as a separate case because the selectivity factor (a percentage of the input) lacks the sensitivity to evaluate this case accurately. This case is one of the commonly defined queries of a database. An example of this type of select would be when we needed to retrieve the phone number of an employee given the employee's name.

The two different cases of the type of results for the select will now be used to develop equations to predict the performance of different hardware configurations and retrieval algorithms. The performance parameters are listed in Table 2. These parameters are based upon the parameters given by DeWitt and Hawthorn [33].

cause these parameters provide a common basis with which to evaluate database machine performance. These parameters will be used for the evaluation of all algorithms and hardware configurations.

The different cases to be developed for each hardware configuration are:

- Case a. Unordered-Unindexed-FT
- Case b. Unordered-Unindexed-MT
- Case c. Unordered-Indexed-FT
- Case d. Unordered-Indexed-MT
- Case e. Ordered-Unindexed-FT
- Case f. Ordered-Unindexed-MT
- Case g. Ordered-Indexed-FT
- Case h. Ordered-Indexed-MT

The first hardware configuration to be examined is the Single Processor- Single Disk cases for the select operator.

### *3.2.1 Case 1. Select - Single Processor-Single Disk.*

*3.2.1.1 Case 1a. Select - Single Processor-Single Disk - Unordered-Unindexed-FT.* The purpose of the select is to provide the tuples that satisfy the given condition. When the relation has been stored in a random manner, the processing of the select requires the relation to be read and scanned by the processor. If the relation was stored in sorted order but the conditions of the select used an attribute(s) other than the attribute(s) of the sort key, this unordered condition would also exist.

The first component of the performance equation is to compile the query,  $T_C$ . Next, the first block of the relation must be found on the disk. This requires a disk access,  $T_D$ . Now, the block is read and transferred to the processor,  $T_{D\rightarrow P}$ . The processor now scans the block comparing the select conditions with the proper attributes in the

$f$	- selectivity factor
$d_j$	- duplicate factor occurring in a project
$j_{sf}$	- join selectivity factor
$p$	- total number of processors
$d$	- number of disks
$p_d$	- number of disk processors
$p_b$	- blocks of memory per processor
$b$	- blocks per track on the disk
$T_c$	- time to compile query
$T_m$	- time to send a message to/from back-end
$T_d$	- average disk access time
$T_s$	- seek time of one track on the disk
$T_{rw}$	- block read/write time
$T_{sc}$	- time to scan block
$T_{bt}$	- time to send block to back-end
$T_p$	- time to process block with complex operation, like join
$T_{ind}$	- time to fetch and examine an index page
$R_t$	- number of tuples in relation R
$S_t$	- number of tuples in relation S
$t_{up}$	- number of tuples selected
$R$	- number of blocks in relation R ( $(R_t * r)/B$ )
$S$	- number of blocks in relation S ( $(S_t * s)/B$ )
$B$	- number of bytes per block
$v$	- attribute size
$r$	- tuple size
$s$	- tuple size
$m$	- index size in bytes

Table 2. Performance equation parameters

tuple to determine if the tuple satisfies the select condition. This time to perform the scan of a block,  $T_{sc}$ , is based upon approximately 100 tuples per block. The time to scan the block also includes the time to move any tuple that meets the select condition to an output buffer. Now the processor is ready for another block to process. Since this is the FT case and few tuples are expected in the results, it is not expected that more than one buffer full of results will be collected. This allows the disk head to remain positioned at the position where the relation was found, allowing the retrieval of the next block to consist of only the I/O. This would continue until the head of the disk needs to move to the next track of the disk which will require a seek,  $T_s$ . The number of seeks necessary is determined by dividing the number of blocks in the relation by the number of blocks contained in a track or cylinder depending on if the disk has multiple platters. The final step of this select operation is to do something with the results. If the results are sent back to the user, then the time parameter used is the block transfer time,  $T_{bt}$ . If the results are stored on disk, the time parameter is the combination of a disk access,  $T_d$ , and the time to write the data,  $T_w$ . The complete equation for Case 1a using the scanning method to perform the select, with R blocks in the relation, when the results are stored upon disk is:

### Model S - 1

$$T_s + T_i + (R * T_{sc}) + (((R/b) - 1) * T_s) + (R * T_d) + T_d + T_w \quad 1$$

or when the result are sent to the user,

### Model S - 2

$$T_s + T_i + (R * T_{sc}) + ((R/b) - 1) * T_s + (R * T_d) + T_d \quad 2$$

Note: this assumes the transfer to some sort of peripheral processor because main terminals can not receive and display the results in this time parameter. But the purpose of this is to compare algorithms and structures, not the time to just results on peripheral devices.

The equations formed previously did not consider the fact that the processor may allow double buffering of the input. This means that the one buffer could be receiving data from the disk while the data in the other buffer was being processed. This allows the overlapping of processing and retrieval time. The equation for this is:

### Model S - 3

$$T_c + T_d + T_{io} + \max \left| \begin{array}{c} ((R-1) * T_{sc}) \\ \text{or} \\ (((R/b)-1) * T_s) + ((R-1) * T_{io}) \end{array} \right| + T_{sc} + T_d + T_{io} \quad (3)$$

or

### Model S - 4

$$T_c + T_d + T_{io} + \max \left| \begin{array}{c} ((R-1) * T_{sc}) \\ \text{or} \\ (((R/b)-1) * T_s) + ((R-1) * T_{io}) \end{array} \right| + T_{sc} + T_{ht} \quad (4)$$

depending upon the placement of the output.

The reason no other algorithm was considered was because to use some form of index, an index would have to be created which requires at a minimum the scanning of the relation plus time to build the index. This would require more time than the simple scan to compare the select conditions.

*3.2.1.2 Case 1b. Select - Single Processor-Single Disk - Unordered-Indexed-MT.* The select with larger results is similar to the FT case except that the processing of the results may interfere with the reading of the data from the disk requiring more costly disk accesses. This is exactly two for each block of result written to the disk. One disk access is necessary to procure the proper place for the result block plus one disk access to return the disk head to the proper location.

to retrieve the next block of the input relation. The equation for this includes an arbitrary parameter, called the selectivity factor,  $f$ , that will determine the volume of the results. The equation for this using double buffering follows:

### Model S - 5

$$\begin{array}{c|c|c}
 & ((R - 1) * T_{sc}) & \\
 & \text{or} & \\
 T_c + T_d + T_{io} + \max & (((R * f) - 1) * 2T_d) & + T_{sc} + T_d + T_{io} \quad (5) \\
 & +(((R/b) - 1) * T_s) & \\
 & +([(R - 1) + ((R * f) - 1] * T_{io})) &
 \end{array}$$

This equation may have fewer disk seeks if some of the disk accesses after writing results overlap with a necessary seek. But this is a small segment of the total execution time, so it was retained to insure at least the minimum time required. It should be noted at this time that any performance equation that cannot be truly expressed for all cases will take the conservative view of expressing the worst case rather than the best case.

The performance model for sending the results to the back-end reflect the reduced disk accesses due to no conflict between reading and writing data to the disk. Therefore, the performance model when the results are sent to a back-end is:

### Model S - 6

$$\begin{array}{c|c|c}
 & ((R - 1) * T_{sc}) + (((R * f) - 1) * T_{bt}) & \\
 & \text{or} & \\
 T_c + T_d + T_{io} + \max & (((R/b) - 1) * T_s) + ((R - 1) * T_{io}) & + T_{sc} + T_{bt} \quad (6)
 \end{array}$$

*3.2.1.3 Case 1c. Select - Single Processor-Single Disk - Unordered-Indexed-FT.* The indexed case indicates that an index exists for the attribute necessary for the evaluation of the selection condition. The indexed case becomes very difficult to model when the selection criteria depends upon multiple attributes being

compared to determine if they satisfy the selection condition. If multiple attributes are required to evaluate the selection criteria, then more than one index may have to be used. In this case, the normal method is to build a list of tuple identifiers (TID) for each condition and then combine the TID lists to determine the tuple(s) that satisfy the selection condition.

The indexed case is ideally suited to the situation where only a few tuples must be retrieved. Assuming the index is searched for an equality condition (e.g., find the name of the person with ID of R2D2), the index reduces the amount of data to be scanned and only the few blocks containing the tuples satisfying the condition are retrieved. However, as the number of tuples that satisfy the selection conditions increase, the effectiveness of the index retrieval decreases because of the increased number of blocks to be retrieved to get the necessary tuples, assuming the tuples are randomly distributed. Also, selection conditions other than equality or equality comparison of two attributes could require the entire index to be searched to determine the tuple addresses or require comparison of more than one index which reduces the advantage of using the indexed retrieval.

In the indexed case it is difficult to predict the performance time because it is very data dependent. Therefore, for this evaluation it is assumed that the number of tuples to be retrieved,  $t_{up}$ , is very small (such as 10 or less) and that each tuple resides in a different block. The performance equation has some fixed parameters for retrievals plus the cost of accessing the index. The cost of accessing the index is determined by computing the depth of the logarithmic index. The other option is to scan all of the leaf nodes of the index but this is not appropriate for finding a few tuples that satisfy a given very specific condition.

The first step of the performance equation is to determine the number of leaf nodes in the index and the number of values that can be referenced from an index node. This is shown by:

$$\text{number of values per index or leaf block} = B^{m-n} + in \quad (7)$$

$$\text{leaf blocks} = (R * (B/r)) / (B/(v + in)) \quad (8)$$

The number of values contained in an index block is the number of bytes in a block divided by the size of the attribute being indexed plus the number of bytes used for the index value. The number of values contained in an index block is then used to compute the number of leaf blocks necessary to index the entire relation. The next step is to determine the number of levels of index necessary to index the number of leaf blocks. This is determined by the logarithm of the number of leaf blocks using the number of values indexed by a block as the base of the logarithm.

$$\text{levels of index } (L_I) = \lceil \log_{(B/(v+in))}(R * (B/r)) / (B/(v + in)) \rceil \quad (9)$$

This value then is used to determine the number of blocks that must be read and scanned to reach the appropriate leaf node. From the leaf node the tuple identifier list is compiled. Then the tuples are retrieved. Since it is assumed that the tuples are randomly distributed, it is assumed that each tuple retrieved requires a disk access plus the time to read and scan the block. The time to complete the query includes a final disk access and I/O to write the results back to disk (if results are going to the user substitute  $T_{bt}$  for  $T_d + T_{io}$ ). The equation is:

### **Model S - 7**

$$T_c + ((L_I + 1) * T_{ind}) + ((T_d + T_{io} + T_{sc}) * t_{up}) + T_d + T_{io} \quad (10)$$

Since the number of tuples that satisfy the selection condition is assumed to be a small number of tuples, the results produced are assumed to fit in a single block. If the results are to be sent to the back-end, then the model is:

### **Model S - 8**

$$T_c + ((L_I + 1) * T_{ind}) + ((T_d + T_{io} + T_{sc}) * t_{up}) + T_{bt} \quad (11)$$

*3.2.1.4 Case 1d. Select - Single Processor-Single Disk - Unordered-Indexed-MT.* The indexed case of the selection operation when more than just a few tuples will be retrieved is very interesting because it is very difficult to accurately predict the performance time. The difficulty arises when determining the number of blocks of the index that must be accessed and scanned and then determining the distribution of the identified tuples among the blocks of the relation. Here it was assumed that the index was of a B-tree type with all actual values and their associated TIDs only contained in the leaf nodes. This allows, for the worst case, only the leaf nodes to be retrieved and scanned. This provides a better method of processing than accessing all of the relation. But the index only provides the address of the tuple, it does not provide the tuple. Since the tuples are stored in an unordered fashion, it has to be assumed the tuples needed are in random blocks. Thus, if the tuples are retrieved as identified during the scanning of the leaf nodes, each tuple retrieval would consist of a disk access + block read + scanning the block. The cost of retrieving more than just a few tuple may be more costly than just scanning the entire relation because reading the entire relation utilizes some disk optimization to reduce the disk access time plus random accesses may re-read blocks multiple times. Therefore, for the index to operate somewhat better it is assumed that the tuple identifiers would be gathered and sorted before any tuple retrievals started. The worst case then requires the leaf blocks to be processed and at most each block of the relation accessed once. The advantage of the index is that it may require only a portion of the leaf nodes to be accessed and then after identifying the tuples, require only a few blocks of the relation to be accessed. The performance equation for case 1d (assuming no restrictions on the number of times a block may be re-read to retrieve tuples) is:

#### Model S - 9

$$T_c + ((L_I + 1) * T_{ind}) + [(T_d + T_{io} + T_{sc}) * T_{up}] + ((T_d + T_{io}) * (R * f)) \quad (12)$$

where  $T_{up} = [(B/r) * R] * f$ .

If the results are presented to the user instead of being stored on disk the equation is:

### Model S - 10

$$T_c + ((L_I + 1) * T_{ind}) + [(T_d + T_{io} + T_{sc}) * ((B/r) * R) * f] + (T_{bt} * (R * f)) \quad (13)$$

The previous equations could require some blocks of the relation to be read several times, each time retrieving just a single tuple. Therefore, this method would need to store the addresses and place them in a sorted order to allow an improved retrieval environment. This method would allow each block of the relation to be read once at most, reducing the number of block reads (especially, reducing the reads as the selectivity factor increases). The performance equations providing this optimized technique are:

$$\text{where } T_{up} = (((B/r) * R) * f)$$

### Model S - 11

$$\begin{aligned}
& T_c + ((L_I + 1) * T_{ind}) \\
& + ((T_{up} * in)/B) * \min \left\{ \begin{array}{c|c|c}
T_b + T_d + \max & R * T_{sc} \\ 
\hline
& or & \\
& (R * T_{io}) + ((R/b) * T_s) & \\
\hline
or & & \\
\hline
T_b + T_d + \max & T_{up} * T_{sc} \\ 
\hline
& or & \\
& (T_{up} * T_{io}) + ((T_{up}/b) * T_s) & \\
\hline
\end{array} \right\} \\
& + ((2T_d + T_{io}) * (R * f)) \quad (14)
\end{aligned}$$

If the results are presented to the user instead of being stored on disk the equation is where  $T_{up} = ((B/r) * R) * f$

## Model S - 12

$$\begin{aligned}
 & T_c + ((L_I + 1) * T_{ind}) \\
 & + ((T_{up} * in)/B) * \min \left\{ \begin{array}{l|l}
 T_b + T_d + \max \left\{ \begin{array}{l|l}
 R * T_{sc} & \text{or} \\
 (R * T_{io}) + ((R/b) * T_s) & \end{array} \right. \\
 \text{or} \\
 T_b + T_d + \max \left\{ \begin{array}{l|l}
 T_{up} * T_{sc} & \text{or} \\
 (T_{up} * T_{io}) + ((T_{up}/b) * T_s) & \end{array} \right. \\
 \end{array} \right\} \\
 & + (T_{bt} * (R * f)) \quad (15)
 \end{aligned}$$

### 3.2.1.5 Case 1e. Select - Single Processor-Single Disk - Ordered-Unindexed-FT.

The ordered case means the relation is stored in a sorted order based upon attribute being used for the selection criteria of the query. Ordering the relation does not mean that it can be determined which of the disk blocks to start with. What this means is that the data is sequential. This means that the only method that can be applied for all disk configurations (if blocks on a disk are mapped in the disk index, a binary search could be used but when blocks are linked together, sequential access is necessary) is the simple scanning method described for cases 1a and 1b. But the processing may be terminated before the entire relation is scanned with the ordered relation. This early termination occurs because the sorted order insures that once the attribute value exceeds the selection criteria, no smaller values will be discovered in later blocks. On the average this would suggest that the expected value of the number of blocks to be scanned would be 1/2 the total number of blocks of the relation. The equation for this is the same as case 1a except the number of blocks to be retrieved and scanned is assumed to be  $1/2 * R$ , realizing of course that the worst case would be exactly the same as case 1a. But this does

provide for comparison purposes that the ordered case on the average will perform better and even for the worst case will perform equally well.

The performance equations are:

### Model S - 13

$$T_c + T_d + T_{io} + \max \left| \begin{array}{c} .5(R * T_{sc}) \\ or \\ .5[((R/b) - 1) * T_s] + (.5R * T_{io}) \end{array} \right| + T_d + T_{io} \quad (16)$$

or

### Model S - 14

$$T_c + T_d + T_{io} + \max \left| \begin{array}{c} .5(R * T_{sc}) \\ or \\ .5[((R/b) - 1) * T_s] + (.5R * T_{io}) \end{array} \right| + T_{bt} \quad (17)$$

depending upon the placement of the output.

*3.2.1.6 Case 1f. Select - Single Processor-Single Disk - Ordered-Unindexed-MT.* This case is the same as the previous case in that it only reduces the number of blocks that may have to be processed from case 1b. However, here it is assumed that the expected value to find the starting location of the desired tuples is half of the remaining blocks after the size of the results is subtracted from the size of the relation ( $.5(R - (R * f))$ ). Then, the results must be read and either stored back on the disk as results or sent to the back-end. The first step in this model then determines the number of blocks to be read to find the starting point of the results and retrieve the results. ( $b_1$ ).

$$b_1 = .5(R - (R * f)) + (R * f) \quad (18)$$

The performance equations using  $b_1$  are:

### Model S - 15

$$T_c + T_d + T_{io} + \max \left| \begin{array}{l} b_1 * T_{sc} \\ \text{or} \\ [((R * f) - 1) * 2T_d] \\ +[((b_1/b) - 1) * T_s] \\ +([(b_1 - 1) + ((R * f) - 1)] * T_{io}) \end{array} \right| + T_i + T_{io} \quad (19)$$

and

### Model S - 16

$$T_c + T_d + T_{io} + \max \left| \begin{array}{l} (b_1 * T_{sc}) + [((R * f) - 1) * T_{bt}] \\ \text{or} \\ [((b_1/b) - 1) * T_s] \\ +((b_1 - 1) * T_{io}) \end{array} \right| + T_{bt} \quad (20)$$

*3.2.1.7 Case 1g. Select - Single Processor-Single Disk - Ordered-Indexed-FT.* This case is similar to case 1c except that now it is not necessary to assume the random distribution of the tuple. Therefore, for the case of retrieving the very few tuples at most the tuples should be contained within two blocks. Thus, modifying the equation from case 1c by replacing the number of TIDs with the maximum of two blocks, produces:

### Model S - 17

$$T_c + ((L_I + 1) * T_{ind}) + ((T_d + T_{io} + T_{sc}) * 2) + T_d + T_{io} \quad (21)$$

and

### Model S - 18

$$T_c + ((L_I + 1) * T_{ind}) + ((T_d + T_{io} + T_{sc}) * 2) + T_b \quad (22)$$

*3.2.1.8 Case 1h. Select - Single Processor-Single Disk - Ordered-Indexed-MT.* This case is similar to case 1d but for case 1d no equation was developed because it was said that for the average case it would perform worse than case 1a and it was very query dependent. However, by adding an index the situation changes greatly. Now, the necessary tuples are grouped and they may be directly accessed through the use of the index. Thus, this case requires only the number of blocks of the relation to be retrieved that are really necessary. This means that its performance should be overall the best.

The actual performance equation includes three segments. The first shows the time to use the index to find the address of the starting block for processing. Next, the time to retrieve and process the blocks of the relation and the last segment shows the time for storing the results on disk. Again, it must be remembered that time for storing the result can be replaced by the time to send the results to some front-end processor for processing to the user. Both equations will be presented to illustrate the difference.

### **Model S - 19**

$$T_c + ((L_I + 1) * T_{ind}) + ((2T_d + 2T_{io} + T_{sc}) * (R * f)) \quad (23)$$

or

### **Model S - 20**

$$T_c + ((L_I + 1) * T_{ind}) + ((T_d + T_{io} + T_{sc}) * (R * f)) + (R * f) * T_{bt} \quad (24)$$

*3.2.2 Case 2. Select - Single Processor-Multiple Disks.* The use of multiple disks allows more options in the storage and retrieval of the relations. There are now three different ways that data may be stored. The first is to use a round-robin method of assigning new data to the disks. This method insures an even distribution

over the disks used. However, this method provides no sequencing or ordering of the data.

The next method provides some grouping of the data by value but does not guarantee that the even distribution of the data is maintained without periodic maintenance. This method, which will be called the hashed method, hashes each tuple upon some attribute(s) and distributes the data based upon this hash value and the boundaries established for each disk. This is the same concept used for a bucket sort. Then within each disk the data may be stored in an unordered manner or it may be sorted and maintained in ordered fashion.

The final method of storage is the fully ordered method. This method orders the data across all the disks that store the data. This method provides ordered data and allows even distribution over the disks. This allows the first block of each disk to be retrieved and examined to find the disks that contain tuples that meet the selection condition.

The multiple storage units do not provide significant differences in the methods for performing the select operator with a single processor. Therefore, the same basic cases that were examined for the single processor-single disk scenario will be used for exploring the single processor-multiple disks environment.

*3.2.2.1 Case 2a. Select - Single Processor-Multiple Disks - Unordered-Indexed FT.* The use of multiple storage units does not effect the processing necessary to perform the select operation. The processor must still scan the entire relation, selecting the tuples that satisfy the selection condition. The only variance from the single processor-single disk case 1a is that the disk operations (seeks and accesses) may be overlapped, since more than one disk is used. This is caused because the processor can issue the same command, i.e., find first block of relation XY, to all the disks. Then after a disk has transferred all the blocks on the cylinder, another disk can transfer data while the other disk(s) seek the next track. The result is

the overlapped disk operations. This produces the following performance equation (assuming double buffering):

### Model S - 21

$$T_c + T_d + T_{io} + \max \left| \begin{array}{c} (R - 1) * T_{sc} \\ \text{or} \\ ((R - 1)/d) * T_{io} \end{array} \right| + T_{sc} + T_d + T_{io} \quad (25)$$

It must be remembered that this is the case where very limited results are expected. Therefore, the results written back to disk fit in one block and the time to write that block,  $(T_d + T_{io})$ , occurs after the processing is complete. Also, if the results are sent to a back-end, the equation would be:

### Model S - 22

$$T_c + T_d + T_{io} + \max \left| \begin{array}{c} (R - 1) * T_{sc} \\ \text{or} \\ ((R - 1)/d) * T_{io} \end{array} \right| + T_{sc} + T_{bt} \quad (26)$$

*3.2.2.2 Case 2b. Select - Single Processor-Multiple Disks - Unordered-Unindexed MT.* The more inclusive case provides for more extensive results; however, this case produces the same modifications of the single disk case – reduction of disk accesses and seeks – as the just completed FT case. Therefore, the only modification is to reduce the seek and accesses through overlapping them. It is assumed that to store the results a disk access is necessary. If one disk was no longer needed for retrieving the relation, it could be dedicated to storing results and thus reduce time to access the proper location on the disk. The following equation distributes the results to free disks to reduce the disk processing time

### Model S - 23

$$T_c + T_d + T_{io} + \max \left| \begin{array}{c} (R - 1) * T_{sc} \\ \text{or} \\ + [( (R - 1)/d ) + (((R * f) - 1)/d)] * T_{io} \end{array} \right| + T_{sc} + T_d + T_{io} \quad (27)$$

If the results were sent directly to the user for processing the results would be:

### Model S - 24

$$T_c + T_d + T_{io} + \max \left( \begin{array}{l} (((R-1)/d) * T_{sc}) \\ + ((R*f)-1) * T_{bt} \\ \text{or} \\ (R-1) * T_{io} \end{array} \right) + T_{bt} \quad (28)$$

*3.2.2.3 Case 2c. Select - Single Processor-Multiple Disks - Unordered-Indexed FT.* The definition of indexed lacks some clarity when used in the multiple disks case. The question becomes what is indexed? Is it an index of the data contained on each disk with the index stored on the disk it indexes or is it a central index that tells only on which disk a value is contained or is it a centralized complete index that provides a complete reference for each tuple, including disk, track, and block? The first type of index, a localized index, causes the processing of the select to be a series of single disk retrievals. This case will not be considered here because it can be easily modeled using the equation from case 1c in a repeated fashion for the number of disks involved.

The second case, a centralized index with limited index, provides no more information than could be determined using the hashing method of storing the data. This case will be examined in a later section. Therefore, here it is assumed that the indexed case means a centralized index, providing complete references for the location of each tuple.

The concept of a centralized index is that it is located in one place. Therefore, it is assumed that a centralized index is located in one place. The effect of this is that the processing of the index is not improved by having multiple data stores.

But, the processing of an index is always a random-type process, that does not allow preseeking of the necessary information. Therefore, the performance this case most closely resembles is the previous case for the single disk case.

The last step in processing the select using an index is retrieving the tuples that satisfy the selection condition. For this case, where the result are assumed to be very few tuples, the retrievals do not consume much time. But, for later cases where the number of tuple increases, this processing time of the random tuples retrievals can become significant.

The performance equations for the single processor-multiple disks - unordered-indexed - limited results case are:

### **Model S - 25**

$$T_c + ((L_I + 1) * T_{ind}) + ((T_d + T_{io} + T_{sc}) * t_{up}) + T_d + T_{io} \quad (29)$$

and

### **Model S - 26**

$$T_c + ((L_I + 1) * T_{ind}) + ((T_d + T_{io} + T_{sc}) * t_{up}) + T_{bt} \quad (30)$$

(See case 1c for explanation of  $L_I$ ).

*3.2.2.4 Case 2d. Select - Single Processor-Multiple Disks - Unordered-Indexed MT.* The indexed case for multiple disks as explained in the previous section does not present any significant performance increase over the performance of the single disk case because of the index processing and the random retrieval of the tuples. Therefore, the equation developed for case 1d is also valid for this case. The one alternative possible with multiple disks is to process the index and storing all of the TIDs satisfying the selection condition. Then, sort the TIDs and retrieve the necessary blocks as shown in Case 1d. Considering that the seeks to different

disks may be overlapped for improved processing, the models are: where  $t_{up} = \lceil (B/r) * R \rceil * f$

#### Model S - 27

$$T_c + ((L_I + 1) * T_{ind}) + ((t_{up} * in)/B) * \min \left| \begin{array}{c} T_b + T_d + (R * T_{io}) \\ \text{or} \\ T_b + T_d + (t_{up} * T_{io}) \end{array} \right| + ((T_d + T_{io}) * (R * f)) \quad (31)$$

and

#### Model S - 28

$$T_c + ((L_I + 1) * T_{ind}) + ((t_{up} * in)/B) * \min \left| \begin{array}{c} T_b + T_d + (R * T_{io}) \\ \text{or} \\ T_b + T_d + (t_{up} * T_{io}) \\ + T_{bt} * (R * f) \end{array} \right|$$

However, even using the optimized technique of ordering the relations, if the number of TIDs approaches the number of blocks of the relation, the number of appropriate tuples will exceed the simpler scan every tuple with

##### 3.2.2.5 Case 2e. Select - Single Processor

*Unindexed FT.* The ordered case means that the tuples are read in order based upon the value of the attribute used in the selection condition. However, it is assumed that the tuples are read in a sequential manner from all the blocks of the relation. Thus, if there are all the blocks for the relation, the time required to read the first  $B$  blocks is  $T_b + T_d + (R * T_{io})$ . The time required to read the remaining  $(n - B)$  blocks is  $((n - B) * T_{bt})$ .

RD-R189 844

A METHODOLOGY BASED ON ANALYTICAL MODELING FOR THE  
DESIGN OF PARALLEL AND.. (U) AIR FORCE INST OF TECH  
WRIGHT-PATTERSON AFB OH SCHOOL OF ENGI.. T G KERNIS

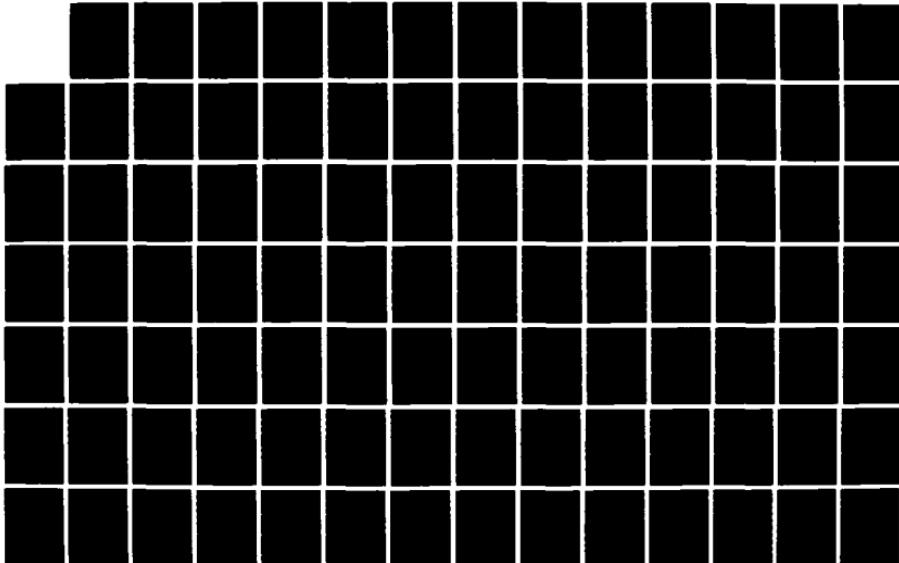
2/4

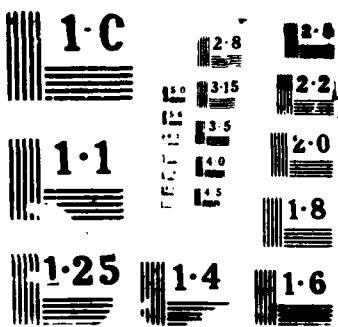
UNCLASSIFIED

DEC 87 AFIT/DS/ENG/87-1

F/8 12/6

ML





This performance equation uses the assumption that the blocks within a disk may have to be processed sequentially but that each disk can be accessed independently. This means that to determine the correct disk to start processing, each disk could be sampled to provide the smallest value contained upon that disk. If the disk ordering is known (all the values on disk A are smaller than on disk B, etc.), then the average case would require one-half the disks to retrieve one block to determine the correct disk to start processing.

The processing of the data from the disks is done in an ordered manner. This does require the sequential processing of the data from the disks, not allowing overlapping of the disk seeks and disk accesses as was done in cases 2a and 2b. But, it is assumed that any results would be stored on a different disk than the disk currently providing the original relation. This does reduce some of the conflicting disk accessing. Therefore, the equation has three significant parts: the sampling of the disks,  $.5d * [T_d + T_{io} + T_{sc}]$  (assuming on the average only .5 of the disk must be accessed to find the correct position to begin); the processing of the necessary data + any seeks and disk accesses necessary including the blocks from the disk necessary to reach the correct block on the disk; and the storing of the results. The finding of the proper place on the disk could require reading all of the blocks from the disk. However, again the expect value will be used to approximate the number of blocks to be read as one-half the blocks stored on the disk plus the number of blocks that contain the tuples that satisfy the selection condition. For the FT case, the number of tuples is so small, it is assumed retrieving the necessary tuples causes one additional block read and scan. If it is assumed that there is double buffering the processing of the data and the retrieval and storing of the results can be overlapped. The resulting equation is:

#### Model S - 29

$$T_c + T_d + T_{io} + \max \left| \begin{array}{l} .5d * T_{sc} \text{ or} \\ T_d + T_{io} \end{array} \right| + T_d + T_{io}$$

$$+ \max \left| \begin{array}{c} .5(R/d) * T_{sc} \\ \text{or} \\ (.5(R/d) * T_{io}) + ((.5(R/d)/b) * T_s) \end{array} \right| + T_{sc} + T_d + T_{io} \quad (33)$$

The performance equation first expresses the search of the disk. Then the first block from the disk is read,  $T_d + T_{io}$ . Next, the double buffering allows the processor to start scanning the block as more blocks are retrieved. In total there are  $.5(R/d)+1$  blocks read and scanned. The final block must be scanned and the results written, to account for the final segment of the equation.

The equation when the results are sent to the back-end is:

#### Model S - 30

$$\begin{aligned} & T_c + T_d + T_{io} + \max \left| \begin{array}{c} .5d * T_{sc} \\ \text{or} \\ T_d + T_{io} \end{array} \right| + T_d + T_{io} \\ & + \max \left| \begin{array}{c} .5(R/d) * T_{sc} \\ \text{or} \\ (.5(R/d) * T_{io}) + ((.5(R/d)/b) * T_s) \end{array} \right| + T_{sc} + T_{bt} \end{aligned} \quad (34)$$

*3.2.2.6 Case 2f. Select - Single Processor-Multiple Disks - Ordered-Unindexed MT.* This case uses the same criteria as the previous case but this time the number of tuples that satisfy it is more indeterminate. Therefore, after the proper disk is found, one-half the blocks are assumed to be retrieved before the correct location is found. Then,  $(R*f)$  blocks are retrieved to provide the tuples that satisfy the select condition. Since there are multiple disks available, it is assumed that the results are stored on another available disk. This allows no additional disk accesses, only disk seeks when a track or cylinder becomes full. The resulting equations are (assuming that  $.5(R/d) + ((R * f) - 1)$  is less than  $R$ ):

### Model S - 31

$$\begin{aligned}
 T_c + T_d + T_{io} \max & \left| \begin{array}{c} .5d * T_{sc} \\ \text{or} \\ .5d * [T_d + T_{io}] \end{array} \right| + T_d + T_{io} \\
 + \max & \left| \begin{array}{c} [.5(R/d) + ((R * f) - 1)] * T_{sc} \\ \text{or} \\ ([(.5(R/d) - 1) + (2 * (R * f) - 1)] * T_{io}) \\ + ((.5(R/d)/b) + ((R * f)/b) * T_s) \end{array} \right| + T_{sc} + T_d + T_{io} \quad (35)
 \end{aligned}$$

and

### Model S - 32

$$\begin{aligned}
 T_c + T_d + T_{io} \max & \left| \begin{array}{c} .5d * T_{sc} \\ \text{or} \\ .5d * [T_d + T_{io}] \end{array} \right| + T_d + T_{io} \\
 + \max & \left| \begin{array}{c} [.5(R/d) + ((R * f) - 1)] * T_{sc} \\ + ((R * f) - 1) * T_{bt} \\ \text{or} \\ ([(.5(R/d) - 1) + ((R * f) - 1)] * T_{io}) \\ + ((.5(R/d)/b) + ((R * f)/b) * T_s) \end{array} \right| + T_{sc} + T_{bt} \quad (36)
 \end{aligned}$$

*3.2.2.7 Case 2g. Select - Single Processor-Multiple Disks - Ordered-Indexed FT.* The index takes the previous cases, 2e and 2f, and reduces the search time. Depending upon the level of the index, the index could point to the correct disk or it could point to the correct block upon the disk. The more detailed the index, the more processing time would be required to retrieve and examine the index. Therefore, there is a trade-off between the complete index that directs the reference to the correct block versus the partial index that references only the correct disk. In either case, the ordering of the relation in storage means that the index is

used only to find the initial processing point and all of the tuples to be selected are grouped together at this point. Remembering that if the ordering of the relation is not upon the necessary attribute(s) for the selection criteria, the unordered case must be used.

The performance equation assumes that a complete index is used. This replaces the disk searching of the previous cases with the index processing and eliminates the searching within the disk for the proper block of information. The performance equations reflect this (assuming two blocks contain all the desired tuples). The equations are:

### **Model S - 33**

$$T_c + ((L_I + 1) * T_{ind}) + 2 * (T_d + T_{io} + T_{sc}) + T_d + T_{io} \quad (37)$$

and

### **Model S - 34**

$$T_c + ((L_I + 1) * T_{ind}) + 2 * (T_d + T_{io} + T_{sc}) + T_{bt} \quad (38)$$

(See case 1c for explanation of  $L_I$ ).

*3.2.2.8 Case 2h. Select - Single Processor-Multiple Disks - Ordered-Indexed MT.* This case is very similar to the previous case only this time the results can not be assumed to be all located within a single block. Therefore, the performance equation has been modified to reflect the time to read the necessary blocks from the disk and the writing of the results to a different disk.

### **Model S - 35**

$$T_c + ((L_I + 1) * T_{ind}) + T_d + T_{io} + \max \left| \begin{array}{l} [((R * f) - 1)] * T_{sc} \\ \text{or} \\ 2([(R * f) - 1] * T_{io}) \\ + (((R * f)/b) * T_s) \end{array} \right| + T_{sc} + T_d + T_{io} \quad (39)$$

If the results are to be sent directly to the user the processing time must include the transfer time.

### Model S - 36

$$T_c + ((L_I + 1) * T_{ind}) + T_d + T_{io} + \max \left| \begin{array}{l} [((R * f) - 1)] * (T_{sc} + T_{bt}) \\ \text{or} \\ [(((R * f) - 1)] * T_{io} \\ + (((R * f)/b) * T_s) \end{array} \right| + T_{sc} + T_{bt} \quad (40)$$

(See case 1c for explanation of  $L_I$ ).

*3.2.3 Case 3. Select - Multiple Processors-Single Disk.* The multiple processor-single disk case is not often referred to because it is normally indicated that the disk is the slowest part of the system [11]. However, the multiple processors with large main memories present the opportunity to use more pipelined processing that may eliminate the repeated writing of temporary results back to the disk. This method of pipelining, if it proves successful, could possibly be applied to cases when multiple disks were used in conjunction with multiple processors. Although cost has not been directly referenced here, it must always be remembered that the ideal system can probably never be implemented because reality either makes the cost too much or there is some physical impossibility.

The main question for the processing of the select operator with many processors and a single data store is "How many processors can be gainfully employed?" Since the select is a rather simple operator that requires very little processing time and a lot of data retrieval time, there is not a need to pipeline any processing.

*3.2.3.1 Case 3a. Select - Multiple Processors-Single Disk - Unordered-Indexed FT.* The multiple processors select is done by having each processor scan a portion of the data. This means that the time to scan the data is  $1/p$  times the time for one processor to scan the relation to find the tuples that satisfy the selection

criteria. The performance equation reflects that the processing time may not be the dominant factor in the execution of the select operator by several processors that share a single data store. The disk controller is assumed to be smart enough to handle requests for the next block of a file even though the request may be coming from several processors. The resulting performance model is:

### Model S - 37

$$T_c + T_m + T_d + T_{io} + \max \left| \begin{array}{c} ((R/p) * T_{sc}) \\ or \\ [((R/b) - 1) * T_s] + ((R - 1) * T_{io}) \end{array} \right| \quad (41)$$

$$+ T_d + (p * T_{io}) + ((p/b) * T_s)$$

This method shows several blocks ( $p$ ) of results being stored on disk. However, this is not the true amount of results. By definition, it was said there would be no more than one block of results. The problem results from each processor having to transfer the smallest unit of transfer, a block, to the disk. The disk cannot be expected to be smart enough to be able to combine the partial blocks of results from each processor and eliminate the unused portion of each block passed to it. Therefore, this result needs to be modified to model the time when the results are sent to a single processor and this processor combines the results to form the final results which are assumed for this case to all fit in one block. This performance equation is:

### Model S - 38

$$T_c + T_m + T_d + T_{io} + \max \left| \begin{array}{c} ((R/p) * T_{sc}) \\ or \\ [((R/b) - 1) * T_s] + ((R - 1) * T_{io}) \end{array} \right| \quad (42)$$

$$+ ((p - 1) * (T_{bt} + T_{sc})) + T_d + T_{io}$$

The results show all the processors sending their blocks to a single processor and this processor combining the data,  $((p - 1) * (T_{bt} + T_{sc}))$ , and the final step of storing the one block of results on the disk,  $T_d + T_{io}$ . This method lacks efficiency because of the chokepoint of retrieving the data from the single disk and then having to recombine the data at a single point for storage. However, this model may provide a better performance if the results are to be sent directly to the user. Since sending the results to the user always requires the results to be collected at a single point, the combination of the partial block does not present a problem. Therefore, sending the results to a back-end produces the following performance equation:

### Model S - 39

$$T_c + T_m + T_d + T_{io} + \max \left| \begin{array}{c} ((R/p) * T_{sc}) \\ \text{or} \\ [((R/b) - 1) * T_s] + ((R - 1) * T_{io}) \end{array} \right| + (p * T_{bt}) \quad (43)$$

*3.2.3.2 Case 3b. Select - Multiple Processors-Single Disk - Unordered-Indexed MT.* This case is very similar to the conditions presented in the previous case except that it is not known the results will be so limited. This causes the performance model to have to provide for the results by using the selectivity factor. The results of doing this is partial blocks of results which will be harder to identify and combine. Therefore, the problem becomes one of should the results all be passed to a single processor and be scanned or should partial blocks of results be stored. In the first situation, there is wasted processor time and in the second, there is wasted time to store blank data on the disk and wasted space upon the disk. This problem is magnified when the number of processors is increased. For example, if 6 processors produced 6 blocks of results, with no wasted space, then 12 processors would produce 12 blocks of results with each block being only half used (assuming even distribution of the data). This means that there would be twice as many disk I/Os and half the disk space used would be empty. However, the processing time for each processor should be reduced by a factor of 2 since there were twice as many

processors to scan the data. But, as earlier stated, the slowest part of the multiple processors-single disk system for the select operator is the disk. Therefore, for any situation it is best to identify the weakest part of the system and attempt to improve this before improving a stronger part of the system.

The performance equation for this case is presented three ways: the first shows storing the results on disk without combining - wasting space and disk time; the second equation shows the results being recombined by a single processor before being stored on disk; and the third shows the result being transferred to a back-end to be displayed for the user.

Storing all results:

#### Model S - 40

$$\begin{aligned}
 & T_c + T_m + T_d + T_{io} + \max \left| \begin{array}{l} (((R/p) - 1) * T_{sc}) \\ \text{or} \\ [((R * f) - p) * 2T_d] \\ +[((R/b) - 1) * T_s] \\ +([(R - 1) + ((R * f) - p)] * T_{io}) \\ +T_{sc} + T_d + (T_{io} * p) + ((p/b) * T_s) \end{array} \right. \quad (44)
 \end{aligned}$$

Combining results at end of processing:

#### Model S - 41

$$\begin{aligned}
 & T_c + T_m + T_d + T_{io} + \max \left| \begin{array}{l} ((R/p) * T_{sc}) \\ \text{or} \\ [((R * f) * ((R/p) - 1)/(R/p)) * 2T_d] \\ +[((R/b) - 1) * T_s] + ([(R - 1) \\ +(((R * f) * ((R/p) - 1)/(R/p))) * T_{io}) \\ +((p - 1) * (T_{bt} + T_{sc})) + T_d \\ +[((R * f) - [(R * f) * ((R/p) - 1)/(R/p)]) * T_{io}] \\ + [((R * f) - [(R * f) * ((R/p) - 1)/(R/p)]) / b] * T_s \end{array} \right. \quad (45)
 \end{aligned}$$

Sending results to user:

### Model S - 42

$$T_c + T_m + T_d + T_{io} + \max \left\{ \begin{array}{l} (.5(R/p) * T_{sc}) + \\ [(R * f) * ((R/p) - 1)/(R/p) * T_{bt}] \\ \quad \text{or} \\ [((R/b) - 1) * T_s] \\ + ((R - 1) * T_{io}) \end{array} \right\} + (p * T_{bt}) \quad (46)$$

*3.2.3.3 Case 3c and 3d. Select - Multiple Processors-Single Disk - Unordered-Indexed FT and MT.* The indexed case for multiple processors is no different than the indexed case described when only a single processor was available (case 1c and case 1d). The reason is that the index cannot be evaluated by more than one processor because the index directs which block is to be retrieved next (some parallel processing experiments try to process the index at a lower level in a hit or miss situation but the retrievals from the disk would be greater than the single processor approach for this environment). Therefore, determining the tuple to be retrieved is done by one processor. The next step retrieves the tuples, which can be controlled by a single processor. Therefore, there is no advantage to having multiple processors available for this case.

*3.2.3.4 Case 3e. Select - Multiple Processors-Single Disk - Ordered- Unindexed FT.* The ordered data case without an index is very similar to the un- ordered case without an index. The only difference is that not all of the data must be examined because it can be determined from the ordering when no more tuples will be selected. This on the average says that only half of the data must be processed. Of course the worst case situation results in the same performance as the unordered case, case 3a. The advantage of this case over the unordered data case with multiple processors is that the partial blocks of results are not a problem because the tuples to be selected are grouped together by the ordering of the data. Therefore, only one

processor would find tuples that satisfied the selection condition (or two processors if a block break occurred). Therefore, the performance model does not have to include the potential partial blocks of results but some form of early termination message must be included to allow processors to not request more blocks than necessary. The performance equation when storing the results is:

#### Model S - 43

$$T_c + T_m + T_d + T_{io} + \max \left| \begin{array}{c} (((.5R/p) + 1) * T_{sc}) + (p * T_m) \\ \text{or} \\ [((.5R/b) - 1) * T_s] + (.5R * T_{io}) \end{array} \right| + T_d + T_{io} \quad (47)$$

If the results are sent directly to the user, the performance model is:

#### Model S - 44

$$T_c + T_m + T_d + T_{io} + \max \left| \begin{array}{c} (((.5R/p) + 1) * T_{sc}) + (p * T_m) \\ \text{or} \\ [((.5R/b) - 1) * T_s] + (.5R * T_{io}) \end{array} \right| + T_{dt} \quad (48)$$

*3.2.3.5 Case 3f. Select - Multiple Processors-Single Disk - Ordered- Unindexed MT.* This case is the same as the previous case in that it only reduces the number of blocks that may have to be processed from case 3b. Therefore the equations for this case is:

#### Model S - 45

$$T_c + T_m + T_d + T_{io} + \max \left| \begin{array}{c} (((.5R/p) + (R * f)) * T_{sc}) + (p * T_m) \\ \text{or} \\ [((R * f) - 1) * 2T_d] \\ + [((.5R/b) - 1) * T_s] \\ + [(.5R - 1) + ((R * f) - 1)] * T_{io} \end{array} \right| + T_d + T_{io} \quad (49)$$

When the results are sent directly to the user the results are:

## Model S - 46

$$\begin{array}{c}
 T_c + T_m + T_d + T_{io} + \max \left| \begin{array}{l}
 [((.5R + (R * f))/p) * T_{sc}] + (p * T_m) \\
 [((R * f) - 1) * T_{bt}] \\
 \text{or} \\
 [(((.5R + (R * f))/b) - 1) * T_s] \\
 +[((.5R + (R * f)) - 1) * T_{io}]
 \end{array} \right| + T_{bt} \quad (50)
 \end{array}$$

Both cases assume that  $.5R + (R * f)$  is less than  $R$ . If this is not true, it is constrained to  $R$  since  $R$  is the number of blocks in the entire relation and the maximum number of blocks to be read.

*3.2.3.6 Case 3g and 3h. Select - Multiple Processors-Single Disk - Ordered-Indexed FT and MT.* This case provides an index of the data. Therefore, as explained for case 3c, this case is the same has case 1g and case 1h, respectively. The advantage of the index and ordering of the data is that the processing can start at the exact block that contains some of the desired data and only continues to process until the desired data is completely obtained. This reduces the number of blocks to be processed significantly, making the problem primarily a disk retrieval exercise.

*3.2.4 Case 4. Select - Multiple Processors-Multiple Disks.* The architecture that provides both multiple disks and multiple processors provides both flexibility in the method that the relations are stored and in the manner that relations are processed during retrievals. One difficulty in developing performance equations for multiple processors and multiple disks is providing a general performance equation without tying the results to a specific hardware configuration. More specifically, this involves determining the time it takes for processors to communicate without specifically stating the type of processor communication environment involved. Another point of the system architecture is whether it is assumed that each disk is connected to a single processor or the disk can be used by retrievals by any of the processors or only a group of processors.

The multiple disks provide numerous ways of storing the data on the disks (see Select - Single Processor-Multiple Disks for complete explanation). Therefore, unordered means that the data is evenly distributed over all the disks but the distribution of the data was done only by order of input no grouping of the data has been done by value. The unordered case also applies when the data has been grouped by the value of an attribute(s) and the selection condition of the query involves a different attribute(s). Hashed means that the data has been grouped into buckets by value but within the individual disk the data could be stored in an unordered or sorted sequence. Ordered means that the complete relation has been stored in sorted order. The individual blocks of the sorted relation are stored on the various disks.

*3.2.4.1 Case 4a. Select - Multiple Processors-Multiple Disks - Unordered-  
Unindexed - FT.* The multiple processors-multiple disks case attempts to use parallel processing to improve the efficiency of the select operation. If for each there is an associated processor, this is often referred to as filtering. Filtering the data assumes that the processor can scan the data as quickly as the disk can retrieve data. This concept has been proposed for several special hardware database machines, such as processor per disk head and a processor per track designs [2,33].

The advantage of using data filtering is that the select operation now consists of  $d$ , where  $d$  is the number of disk and associated processors, independent operations. Thus, allowing the select to be completed in  $1/d$  time, assuming equal distribution of the data. However, data filtering defines that each disk have an associated processor. Here it is assumed that it would be nice to have as many processors as disks but that this is not necessary. Therefore, the performance equation recognizes the capability of having more or less processors than the number of disks that data is stored on. Advantages of this are that the number of disks necessary to store that data may increase or decrease without changing the number of processors and that the number of processors may be less than the number of disk which may be the case in a multi-

user database system.

The performance equation, assuming the data has not been ordered or indexed. is evenly distributed across the disks, and remembering this is the few tuples of results case, is:

### Model S - 47

$$T_c + T_m + T_d + T_{io} + \max \left| \begin{array}{c} ((R/p) - 1) * T_{sc} \\ \text{or} \\ [(((R/d)/b) - 1) * T_s] + \\ (((R/d) - 1) * T_{io}) \end{array} \right| + T_{sc} + T_d + T_{io} \quad (51)$$

This shows the results being stored back to disk,  $T_d + T_{io}$ . This assumes that there is a processor for each disk. To modify this for the case where there may be more processors than disks, the number of processors is divided by the number of disks and the results rounded up to the next integer value. This results in the equation:

### Model S - 48

$$T_c + T_m + T_d + T_{io} + \max \left| \begin{array}{c} ((R/p) - 1) * T_{sc} \\ \text{or} \\ [(((R/d)/b) - 1) * T_s] + \\ (((R/d) - 1) * T_{io}) \end{array} \right| + T_{sc} + (p/d) * (T_d + T_{io}) \quad (52)$$

This accounts for the storing of some results from each processor. Notice that the results have not been combined or sent to a centralized location. The following equation accounts for sending the results to the user or a back-end to combine the results.

### Model S - 49

$$T_c + T_m + T_d + T_{io} + \max \left| \begin{array}{l} ((R/p) - 1) * T_{sc} \\ \text{or} \\ [(((R/d)/b) - 1) * T_s] + \\ (((R/d) - 1) * T_{io}) \end{array} \right| + T_{sc} + (p * T_{bt}) \quad (53)$$

Sending the results to the back-end,  $p * T_{bt}$ , implies that a complete block was transferred. This may not be true but the overhead of passing the data requires a disproportionate amount of message passing, so rather than try to redefine the time to pass a partial block of data the complete block time was used.

The equations presented above assume that the data is evenly distributed among the disks. If the data was not evenly distributed the one disk processing time would dominate the others, reducing the advantage of using multiple processors and multiple disks.

*3.2.4.2 Case 4b. Select - Multiple Processors-Multiple Disks - Unordered-Unindexed - MT.* The select case incorporating the concept of more results is very similar to the previous case. Again the data is unordered and evenly distributed across all the disks. The even distribution also assumes a uniform distribution of the values of the attribute(s) used for the selection condition. Then the performance equation including the selection factor are:

### Model S - 50

$$T_c + T_m + T_d + T_{io} + \max \left| \begin{array}{l} ((R/p) - 1) * T_{sc} \\ \text{or} \\ [(((R/d)/b) - 1) * T_s] + \\ (((R/d) - 1) * T_{io}) + \\ [(((R * f) - p)/d) * (2T_d + T_{io})] \\ + T_{sc} + (p/d) * (T_d + T_{io}) \end{array} \right| \quad (54)$$

The equation includes two disk accesses for each block stored since it is assumed that the results could not be stored on the same track as the relation being retrieved for processing. This causes an disk access to store the results and a access to find the correct place to retrieve more of the input relation. Also, each disk receives an equal portion of the results. This fact relies heavily on the fact that the values are randomly distributed. In reality, a true random distribution of tuples satisfying the select conditions would not be likely, resulting in more results at one processor than another. However, this equation does have a margin of time included in the storing of the final results,  $(p/d) * (T_d + T_{io})$ , since the number of blocks of results,  $(R * f)$  is always rounded up to the next integer value.

The sending of the results modifies the equation to give:

### Model S - 51

$$\begin{array}{c|c|c}
 & (((R/p) - 1) * T_{sc}) & \\
 & + (((R * f) - p)/p) * T_{bt} & \\
 \hline
 T_c + T_m + T_d + T_{io} + \max & \text{or} & + T_{sc} + (p * T_{bt}) \quad (55) \\
 & [(((R/d)/b) - 1) * T_s] & \\
 & + (((R/d) - 1) * T_{io}) &
 \end{array}$$

This shows the results being sent to the back-end. The one area of this equation that could be inaccurate is the sending of the results to the back-end. However, it is assumed that the back-end has a buffer to store results so as not to delay other messages and data from coming in and that the overhead of sending the block allows a small portion for procuring the bus (or sending the message on a network without collision or getting the attention of the receiving processor) to send the block.

*3.2.4.3 Case 4c. Select - Multiple Processors-Multiple Disks - Unordered-Indexed - FT.* The indexed case with the data stored on several disks may imply several different data structures. First, it may be assumed that one centralized index is maintained that includes not only the tuple identifier of block and location within

the block but also the disk where the block is located. Second, indexing may mean the relation is evenly distributed over several disk and that for each disk there is an index. And finally, indexing could imply that each tuple has been assigned to a given disk based upon some hash value. This is not true indexing but is a form of finding the tuple based upon value and since in the corresponding previous cases there was nothing that corresponds to the hash-placed case, it will be included here.

The first index type is a centralized complete index. This type of index would have two parts to its processing. The first part of the processing would examine the index to find the disk and tuple identifiers of the tuples that satisfied the selection condition. Then the addresses of the tuple would be used to retrieve the results. The index processing, since it is a single centralized index, would be done by a single processor with a connected disk. The tuple retrieval would require several disks to retrieve the data and possibly several processors to control the retrieval and remove just the desired tuples from the blocks retrieved. This results in the following two part equation.

$$\text{Index processing} - T_c + ((L_I + 1) * T_{ind}) \quad (56)$$

$$L_I = \lceil \log_{(B/(v+in))}((R/d) * (B/r))/(B/(v+in)) \rceil \quad (57)$$

$$\text{Retrieval} - ((T_m + T_d + T_{io} + T_{sc}) * (t_{up}/d)) \quad (58)$$

This provides the results at the processor and then the results must be stored results stored or sent to the back-end. Storing the results produces many partial blocks of information but may be useful in some cases. Adding this to the previous parts of the equation, produces:

### **Model S - 52**

$$T_c + ((L_I + 1) * T_{ind}) + ((T_m + T_d + T_{io} + T_{sc}) * (tup/d)) + T_d + T_{io} \quad (59)$$

The time to store the results is only  $T_d + T_{io}$  because all of the disks can store their results in parallel. However, this equation assumes sending a message for each tuple to be retrieved since there are not many tuples in the results of this case. But, it might be more appropriate for the index processor to group the tuple identifiers by disk and send only one block containing the tuple identifiers (see case 4d below). Either way, it is assumed that there is a processor associated with each disk to receive the message and process the data.

Next, the condition of sending the results back to the back-end must be considered. Again the back-end processes the index and sends messages to the processors to retrieve the tuples that satisfy the conditions and provides the address for the tuples. This changes the time to store the results in the previous equation into the time to transmit the results. Since it was said that there was a processor for each disk, the parameter for the number of processors sending results back is actually the number of disks. This produces:

### Model S - 53

$$T_c + ((L_I + 1) * T_{ir,d}) + ((T_m + T_d + T_{io} + T_{sc}) * (t_{up}/d)) + (d * T_{bt}) \quad (60)$$

The second index condition, each disk has a separate index, will not be examined here because this case is the same as case 1c with the amount of data at each disk reduced to reflect the distributed data. The final index condition was the hash type distribution of the data.

The hashed distribution of the data does insure that the data is evenly distributed among the disks as was previously described. But for simplification purposes, it will be assumed that the data is evenly distributed over all the disks here.

The hashed distribution means that the selection condition value can be hashed. This produces a value which is used to direct the processing to a specific disk for retrieval. However, this does not utilize all the processors and disks during the retrieval, instead forcing a single disk to be used for the retrieval. Since the data on the

disk is assumed to be unordered, the performance equation for this is very similar to the previous case for a single processor - single disk, only with the amount of data to be searched reduced. The hashing of the selection value is assumed to be part of the query compilation. Then the time to pass a message to a processor provides for the instruction for the processor associated with the disk that contains the appropriate tuples. The resulting equation is:

#### Model S - 54

$$T_c + T_m + T_d + T_{io} + \max \left| \begin{array}{c} ((R/d) - 1) * T_{sc} \\ \text{or} \\ [(((R/d)/b) - 1) * T_s] \\ + (((R/d) - 1) * T_{io}) \end{array} \right| + T_{sc} + T_d + T_{io} \quad (61)$$

when storing the results and

#### Model S - 55

$$T_c + T_m + T_d + T_{io} + \max \left| \begin{array}{c} ((R/d) - 1) * T_{sc} \\ \text{or} \\ [(((R/d)/b) - 1) * T_s] \\ + (((R/d) - 1) * T_{io}) \end{array} \right| + T_{sc} + T_{bt} \quad (62)$$

when sending the result to the back-end.

*3.2.4.4 Case 4d. Select - Multiple Processors-Multiple Disks - Unordered-Indexed - MT.* The conditions are the same when the results are expected to be greater. There are still the different index types to be considered. The first, the centralized index results in the following equations for storing the results and sending the results to the back-end, respectively:

#### Model S - 56

$$\begin{aligned} & T_c + ((L_I + 1) * T_{ind}) + (d * T_{bt}) \\ & + ((T_d + T_{io} + T_{sc}) * ((R * f)/d)) + T_f + T_{io} \end{aligned} \quad (63)$$

and

### Model S - 57

$$T_c + ((L_I + 1) * T_{ind}) + (d * T_{bt}) \\ + ((T_d + T_{io} + T_{sc}) * ((R * f)/d)) + (d * T_{bt}) \quad (64)$$

The second index condition considers the index to consist of individual indices for each disk. This condition reduces the retrieval to the concurrent retrieval from several indexed disks by single processors. This means that the retrieval equations for the individually indexed disks are:

$$L_I = \lceil \log_{(B/(v+in))}((R/d) * (B/r))/(B/(v+in)) \rceil \quad (65)$$

When results are stored on disk

### Model S - 58

$$T_c + T_m + ((L_I + 1) * T_{ind}) \\ + [(T_d + T_{io} + T_{sc}) * ((B/r) * (R/d)) * f] \\ + ((T_d + T_{io}) * ((R/d) * f)) \quad (66)$$

when sending the results to the back-end processor

### Model S - 59

$$T_c + T_m + ((L_I + 1) * T_{ind}) \\ + [(T_d + T_{io} + T_{sc}) * ((B/r) * (R/d)) * f] \\ + (T_{bt} * ((R/d) * f)) \quad (67)$$

The third condition uses the hashing distribution of the data. This results in a single disk containing all of the tuples that satisfy the selection condition. The portion of the relation stored upon that disk is then scanned to determine the exact tuple that satisfy the condition. The resulting performance equation for storing the results back upon the disk for later use is:

Model S - 60

$$\begin{aligned}
 & ((R/d) - 1) * T_{sc} \\
 & \text{or} \\
 T_c + T_m + T_d + T_{io} + \max & [(((R/d)/b) - 1) * T_s] + \\
 & [((((R/d) - 1) + ((R * f) - 1)) * T_{io}) \\
 & +[((R * f) - 1) * (2 * T_d)] \\
 & + T_{sc} + T_d + T_{io}
 \end{aligned} \tag{68}$$

and when returning the results to the back-end the performance equation is:

Model S - 61

$$\begin{array}{c}
 ((R/d) - 1) * T_{sc} \\
 + [( (R * f) - 1) * T_{bt}] \\
 or \\
 [(((R/d)/b) - 1) * T_s] + \\
 (((R/d) - 1) * T_{io}) \\
 + T_{sc} + T_{bt}
 \end{array} \quad (69)$$

**3.2.4.5 Case 4e. Select - Multiple Processors-Multiple Disks - Ordered-  
Unindexed - FT.** The ordering of data for multiple disks means that the relation is completely ordered and partitioned to be stored on disk. Any insertions must find the correct location and then inserted in the correct location. Since the entire relation is sorted, it is assumed that a centralized map of the storage structure exists. This map would contain the disks that store portions of the relation and the attribute(s) value (that was used to order the relation) of the last tuple of each disk. Using the map the retrieval could be directed towards the proper disk to start processing. Also, if the tuples that satisfied the selection condition extended onto another disk this could be recognized from the map allowing that retrieval to occur concurrently. The performance equation assumes the worst case that all the desired tuples are stored on a single disk. Then it is assumed that one-half of the portion of the relation

stored on the disk must be scanned before the correct starting tuple is found. This produces the following performance equations. The first stores the results on disk for later processing and the second passes the results to the back-end. Since this is the FT case the results are less than one block of data. The equations are:

### Model S - 62

$$T_c + T_{ind} + T_m + T_d + T_{io} + T_{sc} + \max \left| \begin{array}{c} ((.5(R/d) + 1)/p) * T_{sc} \\ or \\ .5(R/d) * T_{io} + \\ [.5(R/d)/b] * T_s \end{array} \right| + T_d + T_{io} \quad (70)$$

and

### Model S - 63

$$T_c + T_{ind} + T_m + T_d + T_{io} + \max \left| \begin{array}{c} ((.5(R/d) + 1)/p) * T_{sc} \\ or \\ .5(R/d) * T_{io} + \\ [.5(R/d)/b] * T_s \end{array} \right| + T_{bt} \quad (71)$$

*3.2.4.6 Case 4f. Select - Multiple Processors-Multiple Disks - Ordered-Indexed - MT.* The MT case provides the probability of more than one block of results, causing the search of the relation to provide parameters to account for this variable amount of data to handle. The parameter,  $R * f$ , provides the total amount of information selected from the relation. However, in this case it is known that  $(R * f)$  is not evenly distributed among all of the disks because of the ordering of the relation. The ordering of the data provides all the results sequentially once the starting position has been found. The performance consideration for this case is if this sequential retrieval of data crosses disk boundaries. If the retrieval crosses disk boundaries, the retrievals may be accomplished concurrently. Therefore, the parameter,  $(R * f)$ , must be evaluated to see if it is greater than  $.5(R/d)$ . It is

compared to  $.5(R/d)$  because half of the disk is assumed to be read to find the correct starting position of the results. If the results,  $(R * f)$ , are greater than  $.5(R/d)$ , the results are retrieved on more than one disk and the entire portion stored on one disk must be retrieved ( $.5(R/d)$  to find starting position and  $.5(R/d)$  of results). The equations then reflect the two conditions:

If  $(R * f) > .5(R/d)$

**Model S - 64 and**

**Model S - 65**

$$\begin{aligned}
 & ((R/d)/p) * T_{sc} \\
 & \text{or} \\
 T_c + T_{ind} + T_m + T_d + T_{io} + \max & [((R/d) - 1) + (.5(R/d) * T_{io})] \\
 & +[((R/d)/b) - 1] * T_s \\
 & + [.5(R/d) * 2T_d] \\
 & + T_d + T_{io}
 \end{aligned} \tag{72}$$

and

$$\begin{aligned}
 & (((R/d)/p) * T_{sc}) \\
 & + (.5(R/d) * T_{bt}) \\
 T_c + T_{ind} + T_m + T_d + T_{io} + \max & \text{or} \\
 & ((R/d) * T_{io}) + \\
 & [((R/d)/b) - 1] * T_s
 \end{aligned} \tag{73}$$

Else  $(R * f)$  is less than  $.5(R/d)$

$$\begin{aligned}
 & ((.5(R/d)/p) + ((R * f)/p)) * T_{sc} \\
 & \text{or} \\
 T_c + T_{ind} + T_m + T_d + T_{io} + \max & [( (.5(R/d) - 1) + (2 * (R * f) - 1)) * T_{io}] \\
 & +[((.5(R/d)/b) - 1) * T_s] \\
 & + [((R * f) - 1) * 2T_i] \\
 & + T_d + T_{io}
 \end{aligned} \tag{74}$$

and

$$\begin{aligned}
 & (((.5(R/d)/p) + ((R * f)/p)) * T_{sc}) \\
 & + [((R * f) - 1) * T_{bt}] \\
 T_c + T_{ind} + T_m + T_d + T_{io} + \max & \quad \text{or} \quad + T_{bt} \quad (75) \\
 & [( (.5(R/d) + (R * f)) - 1) * T_{io}] + \\
 & [((.5(R/d) + ((R * f) - 1))/b] * T,
 \end{aligned}$$

**3.2.4.7 Case 4g. Select - Multiple Processors-Multiple Disks - Ordered-Indexed - FT.** The indexed ordered case provides the ability to use the index to directly find the starting retrieval point and provides sequential retrievals from that point. The effect of this though is that a single processor does the majority of the work. When only a small number of tuples satisfy the select condition, the effect is not noticed. But in the next section when the results may be lengthy, the effect can slow the process.

The performance equations show that the index is accessed to provide the retrieval point and that one block is retrieved containing the results at that point. The block is scanned to remove the desired tuples and then the results handled as appropriate, stored on disk or sent to the back-end. Again, it is assumed that indexed means a global index of all the partitions of the relation that may be stored on various disks.

The equation for select when the results are stored on the disk:

$$L_I = \lceil \log_{(B/(v+in))}((R/d) * (B/r))/(B/(v+in)) \rceil \quad (76)$$

### Model S - 66

$$T_c + T_m + ((L_I + 1) * T_{ind}) + T_d + T_{io} + T_{sc} + T_d + T_{io} \quad (77)$$

and when the results are returned to the back-end:

### Model S - 67

$$T_c + T_m + ((L_I + 1) * T_{ind}) + T_d + T_{io} + T_{sc} + T_{bt} \quad (78)$$

*3.2.4.8 Case 4h. Select - Multiple Processors-Multiple Disks - Ordered-Indexed - MT.* The MT case assumes that the results may consist of even several blocks of data. Therefore, the indexing and ordering of the relation provide the advantage of a direct starting location but the disk map also needs to be evaluated to see if the selection condition crosses over a disk boundary. If the condition needs data from the successive disk in the sequence, then more than one processor and disk may be concurrently retrieving data. However, the performance equation can not directly model the fact that the results,  $(R * f)$ , may be on three disks. The result is an approximation that evaluates the size of the results and if this is greater than the amount of data stored on any one disk, then the retrieval assumes that reading all of the data from one disk is the dominant time factor and that the other disks and processors can process their piece of the retrieval faster. If the total amount of the results are smaller than the total amount of the relation stored on any one disk, the retrieval assumes that all the results are on the single disk. This provides the following equations, where the first set of equations are when the results are greater than the amount of data than can be stored on a disk:

$$L_I = \lceil \log_{(B/(v+in))}((R/d) * (B/r))/(B/(v + in)) \rceil \quad (79)$$

**Model S - 68 and**

**Model S - 69 If  $(R * f) > (R/d)$**

$$T_c + ((L_I + 1) * T_{ind}) + T_m + T_d + T_{io} \\ + \max \left| \begin{array}{c} (R/d) * T_{sc} \\ \text{or} \\ [(((R/d) - 1) + (R/d) - 1) * T_{io}] + \\ [(R/d) * 2T_d] \end{array} \right| + T_d + T_{io} \quad (80)$$

*(there are no disk seeks because the writing the results forces a disk access each time requiring the repositioning of the disk head)*

and sending the results to the user:

$$T_c + ((L_I + 1) * T_{ind}) + T_m + T_d + T_{io} + \max \left| \begin{array}{c} ((R/d) * T_{sc}) \\ + (((R/d) - 1) * T_{bt}) \\ or \\ ((R/d) * T_{io}) + \\ [((R/d)/b) - 1] * T_s \end{array} \right| \div T_{bt} \quad (81)$$

Else  $(R * f) < (R/d)$

$$T_c + ((L_I + 1) * T_{ind}) + T_m + T_d + T_{io} + \max \left| \begin{array}{c} ((R * f) - 1) * T_{sc} \\ or \\ [2 * ((R * f) - 1) * T_{io}] \\ +[((R * f) - 1) * 2T_d] \end{array} \right| + T_d + T_{io} \quad (82)$$

and

$$T_c + ((L_I + 1) * T_{ind}) + T_m + T_d + T_{io} + \max \left| \begin{array}{c} [(R * f) * T_{sc}] \\ +[((R * f) - 1) * T_{bt}] \\ or \\ [((R * f) - 1) * T_{io}] + \\ [(((R * f) - 1)/b) * T_s] \end{array} \right| + T_{bt} \quad (83)$$

**3.2.5 Summary.** All the models presented assume some structure to the input relation. If the input relation has no structure (the data is unordered and unindexed), the most efficient select algorithm is the scan of the entire relation. Any of the other select algorithms would require the data to be structured before processing, which require the entire relation to be read, and sorted or indexed. This means sorting the data or building the index would require more processing than scanning the relation. Therefore, no preprocessing of the data was considered to place the relation in sorted order or provide an index strictly to improve the performance of the select.

The indexed performance model has a limited efficiency range. The nature of this model depends upon the random nature of the data, requiring a linear growth of disk accesses as the selectivity factor increases. This means that the index algorithm works very effectively when a single tuple satisfies the query but its performance will decrease when as the number of tuples that satisfy the selection criteria and must be retrieved increases.

The third data structure model, the sorted or ordered relation input model, has some potential for improved performance for performing the select. It does not have the direct path to the results capability of the indexed model. Therefore, it must scan the relation the same as the unordered-unindexed case. However, the ordering of the relation allows the processing to stop after the results have been found. Thus scanning only a portion of the input relation. For the worst case, the scanning of the ordered input would be the same as scanning the unordered input.

The ordered-indexed model provides the advantage of a direct path to the results and sequential processing of the results. Therefore, this model improves upon the random retrieval of the indexed case and provides the advantage of the ordered case of being able to determine when no more data satisfies the given condition, allowing the process to terminate. Since the index is used to direct the processing only to the first tuple and sequential processing is used there after, the ordered-indexed case is not affected by the selectivity factor. Therefore, it seems that this should be the data structure used for storing all relations. However, it is impossible to maintain this data structure for all possible queries without duplicating the relation in storage several times to provide the correct sequencing. The final method would be to reorganize the relation before processing each query. This method, as stated earlier, is slower than scanning the relation because the sorting or indexing must read and scan the entire relation before the actual processing of the query could begin. This means that different algorithms will provide the best processing capability in different situations.

The discussion to this point has assumed a single disk-single processor environment. In general, the considerations described also hold for the multiple processors-multiple disks case. When the multiple processor cases are considered, the capability of parallel processing of the data is assumed. This is true for the unindexed cases. The indexed cases assume a single centralized index, restricting the processing to a single processor. This restriction of the processing may mean for given cases the parallel processing capability may provide faster processing than using the index.

The last consideration of the performance of the select processing is determining the "optimum" number of processors and disks in the multiple processors-multiple disks environments. But, there is not a definite answer to the "optimum" number of resources to be used. The only conclusion that can be drawn is that each performance model compares the disk and processor times. This means that the ideal environment would have equal I/O and processing time. This implies that for the select operation the number of disk and number of processors will be approximately equal or the number of disks will be larger than the number of processors. Also, increasing the performance of the processors without some improvement in accessing the data will not provide overall performance improvement.

#### *IV. Modeling the Performance of the Project Operator*

The projection operator extracts attributes from a relation. This is a vertical reduction of the relation versus the horizontal reduction of the selection operation. The projection operator is actually two separate actions - data reduction and duplicate removal. The projection operator reduces the data by eliminating unwanted attributes or columns of the relation. Duplicate removal is necessary because the reduction of data may have eliminated the attribute that provided the uniqueness of that tuple. The performance of the projection operator depends upon how these two actions are implemented.

The data reduction phase of the projection operation can be done by reading and scanning the entire relation, eliminating the unwanted attributes. Since the entire relation must be read and processed, no ordering or indexing of the data can improve the performance of the operation. Therefore, the performance time of the project operator will be evaluated for four cases:

1. Project - Single Processor-Single Disk
2. Project - Single Processor-Multiple Disks
3. Project - Multiple Processors-Single Disk
4. Project - Multiple Processors-Multiple Disks.

Duplicate removal (required only when the key attribute of the relation is eliminated) requires the results to either be sorted to allow comparison of neighbors for duplicates or a process that compares each tuple with every other tuple. If several different fragments of the relation are produced during the data reduction, a sort-merge may be used to remove the duplicates. Even with a single relation, a sort-scan process provides duplicate elimination capability. To provide duplicate elimination without ordering the relation, a complete comparison algorithm can be used.

A complete comparison algorithm requires that each tuple is compared to every other tuple, eliminating a tuple if it matches a previously processed tuple. This method may be improved slightly by using hashing to group the tuples to eliminate some of the comparisons. However, for large relations this process is still slow because it requires so many reads and writes from secondary storage. The problem with hashing (bucket type sort) is the hashing key. Since duplicates are being removed, the key(s) of the relation were removed. Therefore, should a single attribute, a group of attributes, or the entire tuple be used has the hash key? This same consideration also applies to the sorting method. Although this seems to be a simple decision, the software must be smart enough not to pick a small field, i.e., the attribute sex, as the key since it will not discriminate enough to be helpful. The obvious answer of choosing an attribute that is larger, causes the computer to compare larger fields which requires more time for each comparison. However, the key issue does not seem to be as critical for sorting as when using hashing. Therefore, the sort-merge method is the method of choice for duplicate removal [8].

#### 4.1 Case 1. Project - Single Processor-Single Disk

The projection has two situations - no duplicate removal and duplicate removal. In terms of performance analysis, the second situation is just an extension of the first. To perform the first situation and the first phase of any projection operation, the relation must be read and scanned, removing the unwanted attributes as each tuple is scanned. The size of the results is based upon the number and size of the attributes retained. Therefore, the parameter,  $v$ , will be used to express the size (in bytes) of the tuple after the unwanted attributes have been removed. To determine the number of blocks of results, the number of tuples in the relation is computed.  $(R * B)/r$ . Using this and multiplying by the size of each resulting tuple,  $((R * B)/r) * v$ , provides the number of bytes in the results. Dividing this by the number of bytes per block,  $B$ , produces the blocks after the projection operation removes the unwanted attributes.

Thus, the parameter,  $p_f$  - the projection factor, is the number of blocks of results from a project operation. Therefore,  $p_f = (v * ((R * B)/r))/B$ .

The first situation to be modeled is the projection without duplicate removal being necessary. The first equation shows the results being stored on the disk. The second equation models the results being sent back to the backend to be used by the user immediately. Both assume double buffering, to allow overlapping of processing and data retrieval.

Storing results:

### Model P - 1

$$T_c + T_d + T_{io} + \max \left| \begin{array}{c} (R - 1) * T_{sc} \\ \text{or} \\ [((R - 1)/b) * T_s] + \\ [((R - 1) + (p_f - 1)) * T_{io}] \\ + [(p_f - 1) * 2T_d] \end{array} \right| + T_{sc} + T_d + T_{io} \quad (84)$$

Sending results to user:

### Model P - 2

$$T_c + T_d + T_{io} + \max \left| \begin{array}{c} (R - 1) * T_{sc} + \\ [(p_f - 1) * T_{bt}] \\ \text{or} \\ [((R - 1)/b) * T_s] + \\ [(R - 1) * T_{io}] \end{array} \right| + T_{sc} + T_{bt} \quad (85)$$

The cases presented above presume that the key(s) of the relation were not disturbed. Therefore, no duplicate tuples could be introduced or the user did not want the duplicates removed. The situation where duplicates are introduced and must be removed uses the sort-merge method of duplicate removal. The results will be sorted when a complete block(s) (depending upon the amount of memory in the

processor,  $p_b$ ) of results is produced. The sorted partial results then are stored on disk. When all of the relation has been processed through the initial data reduction, the blocks of partial sorted results are then merged with duplicates being removed at each merge. Bitton fully describes the effects of removing duplicates at each phase of the merge process [8]. This requires the introduction of another parameter that estimates the percentage of duplicates. For simplicity purposes, this parameter will not be introduced here because the time reduced for this consideration is a very inaccurate estimation.

The performance equation has two separate phases. The first performs the data reduction and sorts the partial results. The second phase determines the number of merges necessary and the cost of performing these merges. Then the final results are either stored on disk or sent to the backend.

The sort operation is defined to consist of the same time parameter as a complex block operation,  $T_b$ . This is based upon the consideration presented by Hawthorn and DeWitt that sorting a block requires approximately the same number of comparisons as comparing each tuple of two blocks during a join [33]. However, the sort may need to sort more than one block of data. The sort is not a linear order algorithm, so the time to sort  $p_b$  blocks of data is not  $(p_b * T_b)$ . Instead, if the sort is assumed to be of order  $n \log n$ , the ratio

$$\frac{(t_b * p_b) \log(t_b * p_b)}{t_b \log t_b} = \frac{T_{sort}}{T_b} \quad (86)$$

is used to determine the time,  $T_{sort}$ , necessary to sort  $p_b$  blocks, with  $t_b$  tuples per block. Therefore,  $T_{sort} = T_b * [(t_b * p_b) \log(t_b * p_b)] / (t_b \log t_b)$ .  $T_{sort}$  will be used in all future equations for the reference to the time necessary to sort a block(s).

The performance equation to perform a project with duplicate removal storing the results on the disk is:

### Model P - 3

$$\begin{aligned}
 & T_c + T_d + T_{io} + \max \left( \begin{array}{l} ((R-1) * T_{sc}) + \\ [((p_f/p_b) - 1) * T_{sort}] \end{array} \right. \\
 & \quad \quad \quad \text{or} \quad \quad \quad \left. + T_{sc} + T_{sort} + T_d + T_{io} \right. \\
 & \quad \quad \quad [((R-1)/b) * T_s] + \\
 & \quad \quad \quad [((R-1) + (p_f - 1)) * T_{io}] \\
 & \quad \quad \quad \left. + [(p_f/p_b) - 1] * 2T_d \right) \\
 \\
 & + \log(p_f/p_b) * \max \left( \begin{array}{l} ([(p_f/p_b)/2] - 1) * Tb \\ ((p_f/p_b) - 2) * (2 * (T_d + T_{io})) \\ + Tb + (2 * (T_d + T_{io})) \end{array} \right) \quad (87)
 \end{aligned}$$

This equation stores the final sorted-duplicate removed results back on the disk.  
The following equation sends the final results to the back-end for user processing.

### Model P - 4

$$\begin{aligned}
 & T_c + T_d + T_{io} + \max \left| \begin{array}{l} (R-1) * T_{sc} + \\ [((p_f/p_b) - 1) * T_{sort}] \\ \text{or} \\ [((R-1)/b) * T_s] + \\ [((R-1) + (p_f - 1)) * T_{io}] \\ + [(p_f/p_b) - 1] * 2T_d \end{array} \right| + T_{sc} + T_{sort} + T_d + T_{io} \\
 & + \log((p_f/p_b) - 1) * 2T_d + 2T_{io} + \max \left| \begin{array}{l} [((p_f/p_b)/2) - 1] * Tb \\ \text{or} \\ ((p_f/p_b) - 2) * (2(T_d + T_{io})) \end{array} \right| + Tb + (2(T_d + T_{io})) \\
 & + \left| \begin{array}{l} [((p_f/p_b)/2) - 1] * Tb \\ \text{or} \\ ((p_f/p_b) - 2) * (2 * (T_d + T_{io})) \end{array} \right| + Tb + (2 * T_{bt}) \quad (88)
 \end{aligned}$$

#### 4.2 Case 2. Project - Single Processor-Multiple Disks

The single processor - multiple disk architecture provides the advantage that each block of results that is written back to disk does not have to cause a disk access and allows overlapping seeks and accesses. This provides the possibility for better disk performance over the single disk architecture.

The processing of the project with the multiple disk architecture still has two facets, data reduction and duplicate removal. For the first case where the key of the relation is not disturbed, there is no duplicate removal necessary. The performance equations for storing the results and sending the results to the back-end, respectively, are:

### Model P - 5

$$T_c + T_d + T_{io} + \max \left| \begin{array}{c} (R - 1) * T_{sc} \\ \text{or} \\ [((R - 1) + (p_f - 1)) * T_{io}] \\ + [(p_f - 1) * T_d] \end{array} \right| + T_{sc} + T_d + T_{io} \quad (89)$$

Sending results to user:

### Model P - 6

$$T_c + T_d + T_{io} + \max \left| \begin{array}{c} (R - 1) * T_{sc} \\ [(p_f - 1) * T_{bt}] \\ \text{or} \\ [(R - 1) * T_{io}] \end{array} \right| + T_{sc} + T_{bt} \quad (90)$$

The advantage of using multiple disks, as mentioned earlier is the overlapping of disk seeks and accesses. When the results of the data reduction phase require duplicate removal, the disk accessing becomes more complicated. But it is still assumed that the results can be placed upon a different disk than where the retrievals are coming from. This reduces the retrieval time. However, during the merge phase of the duplicate removal, data can not be written at the same time as it is being sent to the processor because this architecture assumes a single channel between the disk and the processor for data transfer. Therefore, the performance equation allowing a single disk access for storing results is:

### Model P - 7

$$T_c + T_d + T_{io} + \max \left| \begin{array}{c} ((R - 1) * T_{sc}) + \\ [((p_f/p_b) - 1) * T_{sort}] \\ \text{or} \\ [((R - 1) + (p_f - 1)) * T_{io}] \\ + [((p_f/p_b) - 1) * T_i] \end{array} \right| + T_{sc} + T_{sort} + T_d + T_{io}$$

$$+ \log(p_f/p_b) * \begin{cases} T_d + 2T_{io} + \max & \left[ \begin{array}{l} \lceil (p_f/p_b)/2 \rceil - 1 \times Tb \\ ((p_f/p_b) - 2) * (T_d + 2T_{io}) \end{array} \right] \\ + Tb + (T_d + 2T_{io}) \end{cases} \quad (91)$$

This equation stores the final sorted-duplicate removed results back on the disk. The following equation sends the final results to the back-end for user processing.

### Model P - 8

$$\begin{aligned} T_c + T_d + T_{io} + \max & \begin{cases} (R - 1) * T_{sc} + \\ [((p_f/p_b) - 1) * T_{sort}] \\ \quad \text{or} \\ [((R - 1) + (p_f - 1)) * T_{io}] \\ +[((p_f/p_b) - 1) * T_d] \end{cases} \\ + \log((p_f/p_b) - 1) * & \begin{cases} T_d + 2T_{io} + \max & \left[ \begin{array}{l} \lceil (p_f)/2 \rceil - 1 \times Tb \\ ((p_f) - 2) * (T_d + 2T_{io}) \end{array} \right] \\ + Tb + (T_d + 2T_{io}) \end{cases} \\ + \begin{cases} T_d + 2T_{io} + \max & \left[ \begin{array}{l} ([p_f]/2) - 1 \times Tb \\ +((p_f - 2) * T_{bt}) \\ \quad \text{or} \\ (p_f - 2) * (T_d + T_{io}) \end{array} \right] \\ + Tb + (2 * T_{bt}) \end{cases} \end{aligned} \quad (92)$$

#### 4.3 Case 3. Project - Multiple Processors-Single Disk

The multiple processors provide the capability to concurrently process during the data reduction phase. The single disk must provide all the data to the processor and store the results, if that is needed. Therefore, the disk provides the focal point

for the performance if the projection operation does not eliminate the key of the relation. The performance equation when storing the results on the disk is:

### Model P - 9

$$\begin{aligned}
 & T_c + T_d + T_{io} + \max \left| \begin{array}{l} (R - 1) * T_{sc} \\ \text{or} \\ [((R - 1)/b) * T_s] + \\ [((R - 1) + (p_f - p)) * T_{io}] \\ + [(p_f - p) * 2T_d] \end{array} \right. \\
 & \quad + T_{sc} + T_d + (T_{io} * p) + ((p/b) * T_s) \tag{93}
 \end{aligned}$$

Sending results to user:

### Model P - 10

$$\begin{aligned}
 & T_c + T_d + T_{io} + \max \left| \begin{array}{l} (((R/p) - 1) * T_{sc}) + \\ [((p_f/p) - 1) * T_{bt}] \\ \text{or} \\ [((R - 1)/b) * T_s] + \\ [(R - 1) * T_{io}] \end{array} \right. \\
 & \quad + T_{sc} + (p * T_{bt}) \tag{94}
 \end{aligned}$$

The project that removes the key for the relation may introduce duplicate tuples. When the duplicates must be removed, the results from the data reduction are sorted. Then, the sorted segments are merged, removing the duplicates during the merge. With the multiple processors, the purpose is to most gainfully employ all processors. Bitton offers that the best method is to reduce the number of segments to twice the number of processors available [8]. The segments can then be merged by using all the processors, forming a binary tree to complete the merge without returning the data to the disk. This utilization of pipelining depends upon the processors being able to process concurrently during the initial phases to reduce the segments to be merged to twice the number of processors. When there is only

one disk, there is obviously some point where the disk can not fully support the processors. This causes some processors to be idle waiting for disk support. At this point it would seem to be more efficient to begin a pipeline process to utilize the processors. But, determining the number of processors to use in the pipeline is difficult to determine ad hoc. Therefore, it was determined that by reducing the number of segments to be merged to twice the number of processors, the processors can be optimally used and then pipelined to complete the merge without rewriting the data to disk [8]. Until this point is reached the disk will be the determining factor on the performance operation. However, leaving the extra processors idle (as bad as it may seem) may be more efficient than trying to provide the pipeline earlier when the results would still have to be stored on the disk for further processing. Therefore, at some point the processors will be idle waiting for data in the pipeline. By waiting until the final results can be computed in the pipeline, intermediate results will not have to be stored (and it easy to determine how to allocate the processors within the pipeline) and performance is improved.

The performance equation for the multiple processors- single disk architecture when duplicates must be removed is difficult to write in a static format. When the merge processing is being performed it would be more accurate to compute the execution time in a loop because the action of the algorithm is a loop and the results may be reduced at each loop. This was ignored in the previous cases but is a consideration, especially when multiple processors are trying to be supported by the single disk. The equation has three distinct parts: the first reduces the data to the desired attributes and sorts the individual blocks of results, the second part of the equation merges the sorted segments until the number of segments is equal to twice the number of processors, and the third part of the equation merges the segments into the final result through the use of a pipelined tree-organization of processors. The segments to be merged decrease in number as they increase in size during the merge process.

The performance equation when the final results are stored on disk is:

### Model P - 11

$$T_c + T_m + T_d + T_{io} + \max \left| \begin{array}{l} (((R/p) - 1) * T_{sc}) + \\ [((p_f/(p * p_b)) - 1) * T_{sort}] \\ \text{or} \\ [((R - 1)/b) * T_s] + \\ [((R - 1) + (p_f - 1)) * T_{io}] \\ + [(p_f/p_b) - 1) * 2T_d] \end{array} \right| + T_{sc} + T_{sort} + T_d + (p * T_{io}) + ((p/b) * T_s)$$

$$+2T_d + 2T_{io} + \max \left| \begin{array}{l} p_f/2p \log(p_f/2p) * T_m \\ \text{or} \\ (p * [p_f/2p \log(p_f/2p)] - 1) * [(2T_d + 2T_{io}) \\ + (T_d + 2T_{io})] \end{array} \right|$$

$$+(T_d + 2T_{io}) * p + T_m$$

[this phase reduces the number of segments to 2p]

$$+2T_d + 2T_{io} + \max \left| \begin{array}{l} p_f/2 * T_m + (\log p - 1) * T_{bt} \\ \text{or} \\ p_f/2 * [(2T_d + 2T_{io}) + (T_d + 2T_{io})] \end{array} \right|$$

$$+(T_d + 2T_{io}) * p + T_m$$

[This phase reduces the number of segments to the number of processors, this requires the entire relation (in fragments) to be read once and written back to disk. During

*[the reading, the requirement of the numerous blocks requires a disk access for each block read. For the output it is assumed that two blocks of data can be written with each disk access]*

$$\begin{array}{c|c|c}
 & p_f/2 * T_m + (\log p - 1) * T_{bt} & \\
 + 2T_d + 2T_{io} + \max & \text{or} & + (T_d + 2T_{io}) + T_m \\
 & p_f/2 * [(2T_d + 2T_{io}) + (T_d + 2T_{io})] &
 \end{array} \quad (95)$$

*[This phase forms the processors into a binary tree (using  $p - 1$  processors). This allows the remaining merges to be done using a pipeline, eliminating intermediate disk storage. The delay to the top of the tree is  $(\log p - 1) * T_{bt}$ .]*

The same situation occurs when the final results are sent to the backend for user processing except, the final phase will not write the results to disk. However, all of the other intermediate data stores are still required. The performance equation when sending the results to the user and duplicates have to be removed for the single disk-multiple processors is:

#### Model P - 12

$$\begin{array}{c|c|c}
 & (((R/p) - 1) * T_{sc}) + & \\
 & [((p_f/(p * p_b)) - 1) * T_{sort}] & \\
 & \text{or} & \\
 T_c + T_m + T_d + T_{io} + \max & [((R - 1)/b) * T_s] + & \\
 & [((R - 1) + (p_f - 1)) * T_{io}] & \\
 & +[((p_f/p_b) - 1) * 2T_d] & \\
 \\ 
 & + T_{sc} + T_{sort} + T_d + (p * T_{io}) + ((p/b) * T_s) & \\
 \\ 
 & p_f/2p \log(p_f/2p) * T_m & \\
 & \text{or} & \\
 + 2T_d + 2T_{io} + \max & (p * [p_f/2p \log(p_f/2p)] - 1) * [2T_d + 2T_{io}] & \\
 & +(T_d + 2T_{io}) &
 \end{array}$$

$$\begin{aligned}
 & + (T_d + 2T_{io}) * p + T_m \\
 \\
 & + 2T_d + 2T_{io} + \max \left| \begin{array}{c} p_f/2p * T_m + ((p_f - 1) * T_{bt}) \\ \text{or} \\ p_f/2 * [(2T_d + 2T_{io}) + (T_d + 2T_{io})] \end{array} \right| \\
 \\
 & + (T_d + 2T_{io}) * p + T_m \\
 \\
 & + 2T_d + 2T_{io} + \max \left| \begin{array}{c} [(p_f/2) - 1] * T_m \\ + (\log p - 1) * T_{bt} \\ + (p_f - 2) * T_{bt} \\ \text{or} \\ p_f/2 * (2T_d + 2T_{io}) \end{array} \right| + T_m + T_{bt} \quad (96)
 \end{aligned}$$

#### 4.4 Case 4. Project - Multiple Processors-Multiple Disks.

The multiple disk-multiple processor case is very similar to the single disk-multiple processors situation discussed in the previous section. The difference of having multiple disks is that it can be assumed that the data is evenly distributed among the disks and that the disks can equally share the storing and retrieval of data. This causes the performance equations to be adjusted to provide for the multiple data stores by primarily reducing the disk time by 1/number of disks. Having multiple disks also reduces or eliminates the number of disk accesses necessary. However, any arbitrary number of disks may not provide a performance time reduction because of the assumed communication between the disks and processors. If it is assumed that only one processor can communicate with a given disk, the performance time will increase because of the additional communication time to send data to other processors.

The following performance equations assume that there is no communication restriction between the disks and processors. The first equation provides the performance model when no duplicates must be removed and the results are stored on disk (in distributed fragments). The second equation models sending the results to the user.

### Model P - 13

$$\begin{aligned}
 & T_c + T_m + T_d + T_{io} + \max \left| \begin{array}{l} ((R/p) - 1) * T_{sc} \\ \text{or} \\ [(((R/d) - 1)/b) * T_s] + \\ [(((R/d) - 1) + ((p_f - p)/d)) * T_{io}] \\ + [(p_f - p)/d] * 2T_d \end{array} \right| \\
 & + T_{sc} + T_d + (T_{io} * p)/d + ((p/d)/d * T_s) \quad (97)
 \end{aligned}$$

Sending results to user:

### Model P - 14

$$\begin{aligned}
 & T_c + T_m + T_d + T_{io} + \max \left| \begin{array}{l} (((R/p) - 1) * T_{sc}) + \\ [((p_f/p) - 1) * T_{bt}] \\ \text{or} \\ [(((R/d) - 1)/b) * T_s] + \\ [((R/d) - 1) * T_{io}] \end{array} \right| + T_{sc} + (p * T_{bt}) \quad (98)
 \end{aligned}$$

The following equations reflect having introduced duplicates tuple during the data reduction phase of the project. The individual blocks are sorted before they are stored and the sorted segments merged to eliminate duplicates. The following equations provide the performance models for sending the results to the disk to be stored and providing them to the user. The first stores the results on the disks.

### Model P - 15

$$\begin{aligned}
 & T_c + T_m + T_d + T_{io} + \max \left| \begin{array}{l} ((R/p) - 1) * T_{sc} + \\ [((p_f/(p * p_b)) - 1) * T_{sort}] \\ \text{or} \\ [(((R/d) - 1)/b) * T_s] + \\ [(((R/d) - 1) + ((p_f/d) - 1)) * T_{io}] \\ +[((p_f/p_b)/d - 1) * 2T_d] \end{array} \right| \\
 & + T_{sc} + T_{sort} + T_d + ((p/d) * T_{io}) + ((p/d)/b * T_s) \\
 & + 2T_d + 2T_{io} + \max \left| \begin{array}{l} p_f/2p \log(p_f/2p) * T_m \\ \text{or} \\ (p * [p_f/2p \log(p_f/2p)] - 1)/d * [(2T_d + 2T_{io}) \\ + (T_d + 2T_{io})] \end{array} \right| \\
 & + (T_d + 2T_{io}) * (p/d) + T_m \\
 & + 2T_d + 2T_{io} + \max \left| \begin{array}{l} (p_f/2p) * T_m \\ \text{or} \\ (p_f/2)/d * [(2T_d + 2T_{io}) + (T_d + 2T_{io})] \end{array} \right| \\
 & + (T_d + 2T_{io}) * (p/d) + T_m \\
 & + 2T_d + 2T_{io} + \max \left| \begin{array}{l} p_f/2 * T_m + (\log p - 1) * T_{bt} \\ \text{or} \\ (p_f/2)/d * [(2T_d + 2T_{io}) + (T_d + 2T_{io})] \end{array} \right| \\
 & + (T_d + 2T_{io}) + T_m
 \end{aligned} \tag{99}$$

Sending the results to the user:

### Model P - 16

$$\begin{aligned}
 & T_c + T_m + T_d + T_{io} + \max \left| \begin{array}{l} ((R/p) - 1) * T_{sc} + \\ [((p_f/(p * p_b)) - 1) * T_{sort}] \\ \text{or} \\ [(((R/d) - 1)/b) * T_s] + \\ [(((R/d) - 1) + ((p_f/d) - 1)) * T_{io}] \\ +[((p_f/p_b)/d - 1) * 2T_a] \end{array} \right| \\
 & + T_{sc} + T_{sort} + T_d + ((p/d) * T_{io}) + ((p/d)/b * T_s) \\
 & + 2T_d + 2T_{io} + \max \left| \begin{array}{l} p_f/2p \log(p_f/2p) * T_m \\ \text{or} \\ (p * [p_f/2p \log(p_f/2p)] - 1)/d * [(2T_d + 2T_{io}) \\ + (T_d + 2T_{io})] \end{array} \right| \\
 & + (T_d + 2T_{io}) * (p/d) + T_m \\
 & + 2T_d + 2T_{io} + \max \left| \begin{array}{l} (p_f/2p) * T_m \\ \text{or} \\ (p_f/2)/d * [(2T_d + 2T_{io}) + (T_d + 2T_{io})] \end{array} \right| \\
 & + (T_d + 2T_{io}) * (p/d) + T_m \\
 & + 2T_d + 2T_{io} + \max \left| \begin{array}{l} [((p_f/2) - 1) * T_m] \\ +(\log p - 1) * T_{bt} \\ +(p_f - 2) * T_{bt} \\ \text{or} \\ (p_f/2)/d * (2T_d + 2T_{io}) \end{array} \right| + T_m + 2T_{bt} \quad (100)
 \end{aligned}$$

#### *4.5 Summary*

The projection operation models presented all are data structure independent. This means that the data structure does not have an effect on the time required to execute a project. The projection has two phases to completing its operation - data reduction and duplicate removal. The models reflect these two phases for each of the architectures. Therefore, the main performance factors of the project are the architectures ability to utilize parallelism for retrievals and executing the duplicate removal function.

## *V. Modeling the Performance of the Join Operator*

The purpose of the join is to combine two relations, where an attribute(s) of each relation is defined on the same domain and comparison of the attributes (called the join attributes) satisfy the criteria of the join. The join then combines the tuples from each relation to form a tuple of a new relation. The most common join criteria is equality (i.e., where the join attributes have the same value). If a join condition other than equality (i.e., greater than, less than, and not equal) applies, a different situation occurs which forces different processing. Therefore, the cases considered here are only for the equi-join operator.

The join processing has different types of algorithms to consider. All of the algorithms must compare tuples that have equal join attributes. The algorithms either use brute force and compare all tuples or use some form of grouping to reduce the number of tuples to be compared. The join algorithms to be considered here are:

1. nested loop
2. sort-merge
3. index matching
4. hash-based (bucket grouping and joining buckets, only considered for multiple processor or multiple disk situations).

The situation of the join requires that each tuple of relation A is compared with each tuple of relation B. If ordering or grouping insures that all tuples that meet the join criteria are examined, then the complete comparison may be relaxed. It would be nice to have a new improved implementation for the equality join, especially for the parallel processing environment. That does not seem reasonable to expect since there are limited ways to group and compare two relations. However, for various data configurations, different algorithms may provide better performance than an algorithm that is better for a different situation. One consideration is the fact that any database system needs to be able to provide any join condition (other than

equality). This implies that each database system must use the nested-loop algorithm for some join conditions because it insures that it can successfully perform the join for any condition or the product operation.

The different data structures to be considered for the join processing are unordered relations, ordered relations, and indexed relations. All various combinations of the different types of data structures will be considered for each type of architecture and for the different type of join algorithms. The different data situations to be modeled are:

Relation R	Relation S
Case a. Unordered	Unordered
Case b. Ordered	Unordered
Case c. Ordered	Ordered
Case d. Indexed-Unordered	Unordered
Case e. Indexed-Unordered	Ordered
Case f. Indexed-Unordered	Indexed-Unordered
Case g. Indexed-Ordered	Unordered
Case h. Indexed-Ordered	Ordered
Case i. Indexed-Ordered	Indexed-Unordered
Case j. Indexed-Ordered	Indexed-Ordered.

The cases presented do not reflect all possible combinations. However, the two relations may be interchanged, which does provide all possible combinations of different data organizations for the two relations. For some data structures with some algorithms, several cases may be modeled with one model. The first architecture to be explored for join processing is the single processor-single disk.

### 5.1 Case 1. Join - Single Processor-Single Disk

The time parameter to be considered for joining two blocks, by comparing each tuple of the first block with each tuple of the second block, is  $T_b$ . This is the same time parameter that is considered for the time to sort a block of tuples. The first algorithm is the nested-loop algorithm.

### *5.1.1 Nested-Loop.*

*5.1.1.1 Nested-Loop. Cases a through j.* The nested-loop algorithm is the brute force algorithm that will handle any join condition including not equal conditions and the product operation. This algorithm compares each tuple of the first relation with each tuple of the second relation. Ullman provides the criteria for optimizing the nested-loop algorithm by using the smaller relation as the control relation [79]. This means the processor's memory will be filled with blocks of the smaller relation and then single blocks of the larger relation read and compared to the blocks of the smaller relation loaded in memory. By using the smaller relation memory to load the processor memory, the number of total disk accesses is reduced. The nested-loop algorithm does not consider the ordering or indexing of the relation. Therefore, it spans all the cases.

A new parameter will be used in the following performance model. It is the join selectivity factor,  $j_{sf}$ . It provides an estimate of the volume of data produced as results of the join. It has been defined by some researchers as a factor of the product of the number of blocks of the input relations [7]. Since the join is a special case of the product operation, here the join selectivity factor is defined as a factor of the product of the number of tuples in the input relations because the Cartesian product forms a result that is equal to the product of the number of tuples of the input relations. Therefore,  $j_{sf}$  is to be a percentage of the tuples produced by the product of two relations. To convert this to the number of blocks produced, the result tuple size ( $r + s$ ) is divided by the block size. This formulation accounts for the product of the tuples. An example of the effect of not applying the selectivity factor to blocks instead of tuples is when each relation is contained in one block and the selectivity factor ( $j_{sf}$ ) is 20%. If the  $j_{sf}$  is multiplied times the product of the blocks, the result is less than one block. But, the product of the tuples (if each block contains 10 tuples) produces 100 tuples. Then multiplying by the 20%, there are 20 tuples of double the size of the tuples of an individual relation. Thus, producing

4 blocks of output. Therefore, the  $j_{sf}$  must be applied to the product of tuple not product of blocks. The number of blocks produced as results by the join then is expressed as  $j_B$ . Where  $j_B$  is defined by:

$$j_B = [(j_{sf} * ((R * (B/r)) * (S * (B/s)))) / (B / (r + s))] \quad (101)$$

The performance model for the nested loop algorithm joining relations  $R$  and  $S$  (where  $R$  and  $S$  are the number pf blocks in each relation), assuming relation  $R$  is the smaller relation, is:

### Model J - 1

$$\begin{aligned} T_c + & [2T_d + (p_b/b) * T_s + (p_b * T_{io})] * (R/p_b) \\ & + [(R * S * T_b) + (S * (R/p_b))] * T_{io} \\ & + [[(j_{sf} * ((R * (B/r)) * (S * (B/s)))) / (B / (r + s))] * (T_{io} + 2T_d)] \end{aligned} \quad (102)$$

When sending the results to the user:

### Model J - 2

$$\begin{aligned} T_c + & [2T_d + (p_b/b) * T_s + (p_b * T_{io})] * (R/p_b) \\ & + [(R * S * T_b) + (S * (R/p_b))] * T_{io} \\ & + [[(j_{sf} * ((R * (B/r)) * (S * (B/s)))) / (B / (r + s))] * T_{bt}] \end{aligned} \quad (103)$$

The model of the nest-loop join does not provide for double buffering of the input or output because the parameter,  $p_b - 2$ , provides one block for input and one block for output. The result of changing this to  $p_b - 4$  (assuming this many blocks are available) is that the disk accesses for writing results to disk could be overlapped with processing. But, the number of times that the loop is processed would increase because of the decrease of the number of blocks available to hold the smaller relation. The trade-off of buffering input and output or using all the memory for processing has cases where each could be best. The buffering of input and output will not be included here because many times the disk accesses could not be overlapped with the processing because the processors would have finished the processing with a block when it also had output ready, reducing the effect of the buffering.

*5.1.2 Sort-Merge.* The sort-merge algorithm for computing the join orders both input relations and then compares the tuples with a merge operation. This type of processing is said to be the most efficient method of performing the equi-join (including the time to sort the relations) [54].

The sorting or ordering of the relations is accomplished upon the attribute(s) necessary for the join criteria to be evaluated. This means that for cases where the relation is stored in sorted order but sorted on a different attribute than necessary for the join, the relation must be treated as an unordered relation. Also, the situation where the join uses multiple attributes requires the relation to be treated as unordered relations.

*5.1.2.1 Sort-Merge. Cases a, d, f.* These cases provide relations that are unordered. The processing then requires that both relations be sorted and then merged. Therefore, there are three phases to the processing: sort relation R, sort relation S, and merge the two relations. The processing of the sorts first loads the memory of the processor, then the blocks in the processor are sorted and the sorted segment is stored on disk. After all of the blocks are sorted, the sort is completed by merging the sorted segments to form the sorted relation. The processing of the sort is not overlapped with data I/O because the data must all be in memory for processing to begin and the results are not complete until the sort is complete.

The merge processing does allow overlapping of the disk I/O and the processing of the merge (if there are enough processor blocks to allow data buffering). The result of this is to reduce the number of disk accesses that would delay the processing of the merge.

The first phase of the sort-merge join is to sort the relations. Therefore, the first performance model is for sorting a relation. This model requires the number of blocks to be input, allowing this module to be used in many of the following performance models. This model is titled SortSS(X), meaning sort the relation

with X blocks using the single processor-single disk architecture. The equation for SortSS(X) when storing the results on disk is:

### Model J - 3

$$\begin{aligned}
 & T_d + (p_b * T_{io}) + Tsort(p_b, tb) + T_d + (p_b * T_{io}) \\
 & + \max \left| \begin{array}{l} ((X/p_b) - 1) * Tsort(p_b, tb) \\ \text{or} \\ (((X/b) - 1) * T_s) + ((X - p_b) * 2T_{io}) \\ + (((X/p_b) - 1) * 2T_d) \end{array} \right| \quad (104) \\
 & + (\log(X/p_b)) * (2T_d + 2T_{io}) + \max \left| \begin{array}{l} ((X/2) - 1) * T_b \\ \text{or} \\ (X - 2) * (2 * (T_d + T_{io})) \end{array} \right| \\
 & + T_b + (2 * (T_d + T_{io}))
 \end{aligned}$$

The final performance model includes sorting of the two relations, merging the two relations, and either sending the results to the user or saving the results on the disk. The model when saving the results on disk is:

### Model J - 4

$$T_c + (R/p_b) * (2[T_d + ((p_b/b) * T_s) + (p_b * T_{io})] + T_{sort})$$

$$+ \log(R/p_b) * \left\{ 2T_d + 2T_{io} + \max \left| \begin{array}{l} ((R/2) - 1) * T_b \\ \text{or} \\ + 2T_{io} * \\ (R - 2) * 2(T_d + T_{io}) \end{array} \right| \right\} + T_b + T_i$$

$$+ (S/p_b) * (2[T_d + ((p_b/b) * T_s) + (p_b * T_{io})] + T_{sort})$$

$$+ \log(S/p_b) * \left\{ 2T_d + 2T_{io} + \max \left| \begin{array}{l} ((S/2) - 1) * T_b \\ \text{or} \\ (S - 2) * 2(T_d + T_{io}) \end{array} \right| \right\} + T_b + T_d + 2T_{io}$$

$$+ 2T_d + 2T_{io} + \max \left| \begin{array}{l} (((R+S)/2) - 1) * T_b \\ \text{or} \\ ((R+S) - 2) * (T_d + T_{io}) \\ + [j_B - 2] * (T_d + T_{io}) \end{array} \right| + T_b + T_d + 2T_{io}$$

(105)

If the results are sent directly to the user, the following performance model is used:

## Model J - 5

$$T_c + (R/p_b) * (2[T_d + ((p_b/b) * T_s) + (p_b * T_{io})] + T_{sort})$$

$$-\log(R/p_b) * \left\{ 2T_d + 2T_{io} + \max \left| \begin{array}{c} ((R/2)-1) * T_b \\ \text{or} \\ (R-2) * 2(T_d + T_{io}) \end{array} \right| + T_b + T_d + 2T_{io} \right\}$$

$$+(S/p_b) * (2[T_d + ((p_b/b) * T_s) + (p_b * T_{io})] + T_{sort})$$

$$+\log(S/p_b) * \left\{ 2T_d + 2T_{io} + \max \left| \begin{array}{c} ((S/2)-1) * T_b \\ \text{or} \\ (S-2) * 2(T_d + T_{io}) \end{array} \right| + T_b + T_d + 2T_{io} \right\}$$

$$+2T_d + 2T_{io} + \max \left| \begin{array}{c} (((R+S)/2)-1) * T_b \\ +[j_B-2] * T_{bt} \\ \text{or} \\ ((R+S)-2) * (T_d + T_{io}) \end{array} \right| + T_b + 2T_{bt}$$

(106)

*5.1.2.2 Sort-Merge. Cases b, e, g, i.* All of these cases provide one relation that is already ordered using the attribute(s) necessary for evaluation of the join condition. This allows the sort-merge processing to sort the remaining relation and then execute the merge. Obviously, this will provide better performance than the previous cases. The models presented here represent relation R as the relation that is not sorted. But, this is an arbitrary identifier since R can be replaced by any value. Therefore, if S was the relation that was sorted the values of R and S would be interchanged for the computation of the execution time.

The sort-merge with one relation already sorted has two phases: sort the unordered relation and merge the sorted results to allow comparison of tuples to de-

determine if they satisfy the join condition. The first equation represents storing the final results on the disk.

Model J - 6

$$T_s + (R/p_b) * (2[T_d + ((p_b/b) * T_s) + (p_b * T_{io})] + T_{sort})$$

$$+ \log(R/p_b) * \left\{ 2T_d + 2T_{io} + \max \left( \begin{array}{c} ((R/2) - 1) * T_b \\ or \\ (R - 2) * 2(T_d + T_{io}) \end{array} \right) + T_b + T_d + 2T_{io} \right\}$$

$+2T_d + 2T_{io} + \max$	<table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="padding: 10px; width: 50%;"> <math>((R + S)/2) - 1) * T_b</math> </td><td style="padding: 10px; width: 50%;"> <math>or</math> </td></tr> <tr> <td style="padding: 10px;"> <math>((R + S) - 2) * (T_d + T_{io})</math> </td><td style="padding: 10px;"></td></tr> <tr> <td style="padding: 10px;"> <math>+ [j_B - 2] * (T_d + T_{io})</math> </td><td style="padding: 10px;"></td></tr> </table>	$((R + S)/2) - 1) * T_b$	$or$	$((R + S) - 2) * (T_d + T_{io})$		$+ [j_B - 2] * (T_d + T_{io})$		$+ T_b + T_d + 2T_{io}$
$((R + S)/2) - 1) * T_b$	$or$							
$((R + S) - 2) * (T_d + T_{io})$								
$+ [j_B - 2] * (T_d + T_{io})$								

( 107 )

If the results are sent directly to the user, the following performance model is used:

Model J - 7

$$T_c + (R/p_b) * (2[T_d + ((p_b/b) * T_s) + (p_b * T_{io})] + T_{sort})$$

$$+ \log(R/p_b) * \left\{ 2T_d + 2T_{io} + \max \begin{cases} ((R/2) - 1) * T_b \\ (R - 2) * 2(T_d + T_{io}) \end{cases} \text{ or } \begin{cases} + T_b + T_d + 2T_{io} \end{cases} \right\}$$

$$+2T_d + 2T_{io} + \max \left\{ \begin{array}{c} (((R+S)/2) - 1) * T_b \\ + [j_B - 2] * T_{bt} \\ or \\ ((R+S) - 2) * (T_d + T_{io}) \end{array} \right\} + T_b + 2T_{bt}$$

(108)

*5.1.2.3 Sort-Merge. Cases c, h, j.* These cases all present both tuples already sorted on the attribute(s) needed for the join processing. This means the join only needs to merge the two relations to complete the join. Again, it is obvious that this requires less time to complete than the previous two cases of the sort-merge operation because there is less processing to do. The model when the results are stored on the disk is:

#### Model J - 8

$$T_c + 2T_d + 2T_{io} + \max \left| \begin{array}{c} (((R + S)/2) - 1) * T_b \\ \text{or} \\ ((R + S) - 2) * (T_d + T_{io}) \\ + [j_B - 2] * (T_d + T_{io}) \end{array} \right| + T_b + T_d + 2T_{io} \quad (109)$$

When the results are sent to the user, the model is:

#### Model J - 9

$$T_c + 2T_d + 2T_{io} + \max \left| \begin{array}{c} (((R + S)/2) - 1) * T_b \\ + [j_B - 2] * T_{bt} \\ \text{or} \\ ((R + S) - 2) * (T_d + T_{io}) \end{array} \right| + T_b + 2T_{bt} \quad (110)$$

*5.1.3 Indexing.* Indexing uses the concept of mapping the values of the attribute(s) needed for the comparison of the join condition to an index of the location of the tuple on the disk. This type of processing compares the indices for matching values and then retrieves the corresponding tuples to be combined and output. The processing modeled here considers only comparing one set of attributes. If multiple conditions were involved in the join, the index processing would have to produce a separate TID list for each part of the join condition and combine the TID list as the join condition dictated. After the final TID list was computed, then the tuples would be retrieved from disk. Also, the index may be constructed over more than

one attribute to facilitate the processing of the multiple equality conditions in one pass of the index.

This processing works best when the index already exists and the number of matching values is small because this means that a small amount of data must be loaded to make the comparison and very few values have to be retrieved for the output. This is not true if the attribute(s) needed for the comparison consists of a majority of the tuples to be processed.

The use of an index depends upon the index being present or being able to rapidly build an index. There are numerous types of indices and different methods to organize or scan the data to build the index. The type of index considered here is a B-tree type index. This index allows a node of the index to address several lower level nodes. This reduces the height of the tree reducing the number of levels necessary to be processed to retrieve each value. The specific index considered will be a B<sup>+</sup>tree. This means that the actual values with references to the tuples that contain them are only contained in the leave nodes. This means all nodes in the levels above the leaf level are used to provide the path to the leaves. The advantage of keeping all the values in the leaves is that the values can be processed sequentially because the leaves are linked together. This provides the sequential order of the values.

The disadvantage of the index is building it. The factors for building the index are the size of the blocks to contain the index, the size of the attribute(s) used for the index, and the number of tuples to be indexed. The problem of building the index is the number of disk accesses necessary to retrieve the data to be indexed, the number of disk reads to find the proper location in the index and then rewriting the block back to the disk plus writing any blocks of the index that had to be changed to reflect the new condition of the index. This also presents a problem in accurately predicting the number of disk accesses. Therefore, there are three methods explored for building the index: build the index with random order data, build the index after

blocks of the data have been sorted, and entirely sort the data before building the index.

The first method of building an index is to scan a relation recording the value of the attribute that is going to be used to index the relation. As the value is found, a address for the tuple is determined. This address, called a tuple identifier (TID), tells what block and the offset within the block where a tuple begins. This can be used to later retrieve the tuple.

The leaf nodes of the index contain the values and the associated TIDs. Therefore, to build the index with random data, each value is removed from its tuple, its TID determined, the existing index searched to determine the leaf node where the new value and TID should be inserted and the leaf node read, changed, and written. Also, any of the index nodes above the modified leaf node may have to be modified if the new value created a new boundary condition. And, a leaf node may become full and have to be split to create two nodes which causes the index nodes in the tree above to be modified. This means that each insertion must at a minimum read a leaf node and write a leaf node (assuming any index nodes used were retained in the processor memory).

Horowitz and Sahni [39] formalize this to include the splitting of index nodes and the number of accesses necessary for each insertion. The average number of accesses necessary is equal to the levels of the index plus 2 divided by the sum of the number of index values that may be contained in a block divided by 2 minus 1. This is:

$$\text{accesses} = l + (2/[(z/2) - 1]) \quad (111)$$

where  $l = (\log_z(i * tb)/z)$

$i$  = the number of the block being processed

$tb$  = number of tuples per block ( $B/r$ )

$z$  = number of index entries per block ( $B/(v + in)$ )

The total performance of building a index without sorting is:

$$((R/p_b) * T_d) + (R * T_{io}) + (R * (p_b/b) * T_s) + (R * T_{sc}) + (\text{total accesses} * T_{ind}) \quad (112)$$

Total accesses is the sum of the accesses for each block. This means that the accesses for the block being processed depend upon the number of blocks that have already been processed. Therefore,

$$\text{total accesses} = \sum_{i=1}^R [\log_z((i * tb)/z) + (2/((z/2) - 1))] \quad (113)$$

The next method of building the index is to first sort the contents of the processor memory before inserting the values into the index. This reduces the random accesses into the index and allows that, at most, each leaf node is accessed at most once and that the index nodes are at most modified only once. Which reduces the number of accesses per value inserted. The formulation of this model must include the time to sort the contents of the processor memory.

### Model J - 10

$$((R/p_b) * T_d) + (R * T_{io}) + (R * (p_b/b) * T_s) + (R/p_b * T_{sort}) + (\text{total accesses}) \quad (114)$$

where total accesses is:

using  $l_b = ((i - 1) * tb * (p_b/z))$   
 $l_b\_next = (i * tb * (p_b/z))$   
 $tb = \text{number of tuples per block } (B/r)$   
 $z = \text{number of index entries per block } (B/(v + in))$

$$\sum_{j=1}^{R/p_b} \left\{ \begin{array}{l} l_b > (tb * p_b) \\ \text{or} \\ l_b \leq (tb * p_b) \end{array} \right\} \left\{ \begin{array}{l} +((tb * p_b) * (T_{ind} + T_{io} + T_d)) \\ +[l_b.next - l_b] * T_{ind} \\ + \sum_{j=1}^{\log_z l_b.next} [((tb * p_b)/z) + (((1/(z/2) - 1) * tb * p_b)] * T_{ind} \\ +(l_b.next * T_{ind}) + (l_b * (T_d + T_{io})) \\ +[(i * tb * p_b) + (((1/(z/2) - 1) * l_b.next))] * T_{ind} \end{array} \right. \quad (115)$$

where  $Rr = R * ((v + in)/r)$

$$\begin{aligned}
 & ((R/p_b) * T_d) + (R * T_{io}) + (R * (p_b/p) * T_s) + ((Rr/p_b) * T_d) \\
 & +(Rr * T_{io}) + (Rr * (p_b/b) * T_s) + ((R/p_b) * T_{sort}) + [(\log_{p_b-1}(Rr)) * (2 * Rr * (T_d + T_{io}))] \\
 & +((Rr/2) * T_b) + (((R * tb)/z) * T_{ind}) + \sum_{i=0}^{\log_z(R*tb)/z} ((R * tb) * T_{ind})
 \end{aligned}$$

The results show that at times sorting the projection of the relation that will be used to index the relation may be the most efficient method of building the index. But, if the index has to be formed for both relations, the time to build the index would include the time necessary to sort the relations and the time to scan the relations to build the index. This is equal to the time necessary to complete the sort-merge processing of the relations. The index processing would still require the processing of the indices and retrieval of the necessary tuples from the disk. Therefore, the use of indices will not be considered unless at least one of the relations is indexed on the attribute(s) necessary for the join processing.

The use of indices for processing the join also vary if the data is grouped corresponding to the index. If the relation is not ordered corresponding to the index, each tuple to be retrieved must be done individually. This means that for each tuple a disk access and I/O will be required to retrieve the block that contains

the tuple needed. Therefore, the number of disk accesses could exceed the number of blocks in the relation. These are some of the disadvantages of using index processing for executing the join. The next step is to model the performance of using indices to process the join.

The use of indices to complete the join has three phases: build the index, compare the values in the indices to identify the location of the tuples to be contained in the results, and retrieving the tuples and building the results. The first phase of building the indexes may already be accomplished if the data structure includes a index. However, some instances will have indexed relations but the index is not on the correct attribute. This would require a new index to be constructed to complete the processing. Only one method of building an index will be considered. This method reads as much of the relation into the memory as possible, the processor sorts this portion of the relation and then the sorted fragment is used to build the index. This is repeated until the entire relation has been processed.

The next phase of index processing of the join is the comparison of the indices. Here the values are sequentially processed from each index. When the values match, the addresses of the tuples are used to retrieve the tuples for combination. It is assumed that the index allows sequential processing of the values as was discussed previously. If sequential processing of the index was not possible, then the attribute values of one relation would be compared with the values in the index of the other relation. This would require a probe into the index for each value. This would mean several disk accesses for each probe which would eliminate the benefit of having the index for improved processing. This illustrates the purpose of the index: fast retrieval of a single tuple of a relation based upon a given attribute value. The value of the index decreases with every additional tuple that is required to be retrieved because each additional tuple requires a probe the depth of the index which requires additional disk accesses.

The final phase was the retrieval of the identified tuples. For each tuple an

address is provided. This address includes the block number and offset within the block. If the addresses are provided in random order this means that the same block could be repeatedly retrieved to procure different tuples. This again causes excess disk accesses. Therefore, the clustering of the data upon the indexing value provides the capability of ordering the retrievals so that at a maximum each block of the relation would be accessed once. Next, all of the considerations of building the indexes, processing the index, and tuple retrievals will be applied to the various data configurations to provide performance models.

*5.1.3.1 Index. Case a.* Case A provide two relations that have no sequence and no index provided. This requires that an index be constructed for both relations before the index processing can begin. The performance model of building the indexes the relations R and S is:

**Model J - 11**

*Construct Index for R*

*+ Construct Index for S*

The next step is using the index to compare the attribute values to determine which tuples need to be retrieved and combined to complete the join. Since the indexes built provide the ability for sequential processing of the values contained, only the leaf blocks of the index must be accessed. Then the leaf blocks from each index are compared. Since the values are in sequential order this is the same as the merge process. Therefore, the segment of the performance model first provides the disk accesses to read the leaf blocks (a disk access is assumed for each because of the uncertain nature of the order of retrieval and the disk head may move to retrieve blocks of the relation in the interim). Next, the leaf blocks are merged to provide TIDs for the necessary tuples. Since the relations are not stored in sequential order, the retrieval using the TIDs requires a disk read for each TID. Finally, the results must also be stored or sent to the user to complete the performance model.

$R_l$  and  $S_l$  represent the number of leaf nodes in the index for R and S, respectively.  $R_l$  and  $S_l$  are computed by the formula:

(Number of Blocks \* Tuples per Block) \* (Attribute(s) to indexed + Address Size (TID)) / Block Size.

Therefore, this segment of the model storing results is:

### Model J - 12

$$(R_l + S_l) * (T_d + T_{io}) + ((R_l + S_l)/2) * T_t \\ + [(j_{sf} * ((R * (B/r)) * (S * (B/s))))] * 2(T_d + T_{io}) + (j_B * (T_d + T_{io})) \quad (116)$$

Sending the results is:

### Model J - 13

$$(R_l + S_l) * (T_d + T_{io}) + ((R_l + S_l)/2) * T_t + \\ [(j_{sf} * ((R * (B/r)) * (S * (B/s))))] * 2(T_d + T_{io}) + (j_B * T_{bt}) \quad (117)$$

Thus, the total time required to perform the join with indexes is the sum of the two segments – index building plus processing of the indexes to complete the join.

*5.1.3.2 Index. Case b.* This case provides one sorted relation and the other in random order. The processing using indexes here has two options: build indexes for both and process as in the previous case (except the ordered relation would require fewer disk access to retrieve results) or build an index for the random order relation and process the index with the sequential relation.

The ordered relation allows the building of the index to proceed more quickly because the relation because only the last segment of the indexing building is applied. However, the building of the index for the sorted relation requires reading the entire relation plus the time required to insert the attribute values into the index. After the indexes are built, the join processing can begin. During the retrieval of tuples

that satisfy the join condition, the clustering provided in the ordered relation limits the accesses to that relation to one access per block. Whereas, the random sequence of the other relation, may require an access for each tuple retrieved. This means that building the index and accessing the tuples of the ordered relation requires the relation to be read twice plus the time required to build and process the index. If the ordered relation is used without building an index, the relation is read only once and compared to the index of the second relation. This requires the same processing for the join part of the processing but eliminates the time to build the index for the ordered relation. Thus, reducing the overall processing time to complete the join. Therefore, only the situation of building an index for the unordered relation and comparing this with the sorted relation will be modeled. This performance model is:

#### **Model J - 14**

*Build index for S*

$$+(R + S_l) * (T_d + T_{io}) + ((R + S_l)/2) * T_b \\ + [(j_{sf} * ((R * (B/r)) * (S * (B/s))))) * (T_d + T_{io}) + (j_B * (T_d + T_{io}))] \quad (118)$$

Sending the results is:

#### **Model J - 15**

*Build index for S*

$$+(R + S_l) * (T_d + T_{io}) + ((R + S_l)/2) * T_b + \\ [(j_{sf} * ((R * (B/r)) * (S * (B/s))))) * (T_d + T_{io}) + (j_B * T_{bt})] \quad (119)$$

**5.1.3.3 Index. Case c.** The situation where both relations are in sorted order does not provide a good environment for index processing of the join. If index processing of the join is to be accomplished, the first step is to build the indexes. The building of the indexes requires the scanning of the relations, which requires reading

all of the blocks of the relations. This combined with processing time to build the index is approximately equal to the time to perform a merge of the relation to execute the join. Therefore, the execution of the join with the indexes would cause excessive time to be used. Thus, for the case of both relations being ordered and not indexed, the best method is the sort-merge approach and no index performance model will be shown.

*5.1.3.4 Index. Case d.* Case d provides one relation that is indexed and both relations are unordered. The approach to performing the join here is to create the necessary second index and proceed with the join processing of the join (described in case a). The performance model for this is:

#### **Model J - 16**

*Build index for S*

$$+(R_l + S_l) * (T_d + T_{io}) + ((R_l + S_l)/2) * T_b \\ + [(j_{sf} * ((R * (B/r)) * (S * (B/s))))] * 2(T_d + T_{io}) + (j_B * (T_d + T_{io})) \quad (120)$$

Sending the results is:

#### **Model J - 17**

*Build index for S*

$$+(R_l + S_l) * (T_d + T_{io}) + ((R_l + S_l)/2) * T_b + \\ [(j_{sf} * ((R * (B/r)) * (S * (B/s))))] * 2(T_d + T_{io}) + (j_B * T_{bt}) \quad (121)$$

*5.1.3.5 Index. Case e.* This case with one indexed relation and one ordered relation provides a situation similar to case b. However, this case does not require the building of the index for the unordered relation. Thus, this provides the opportunity for improved join processing. The performance model then uses the sorted relation to be compared with the index of the unordered relation. The performance model is:

## Model J - 18

$$(R_l + S) * (T_d + T_{io}) + ((R_l + S)/2) * T_b \\ + [(j_{sf} * ((R * (B/r)) * (S * (B/s))))) * (T_d + T_{io}) + (j_B * (T_d + T_{io}))] \quad (122)$$

Sending the results is:

## Model J - 19

$$(R_l + S) * (T_d + T_{io}) + ((R_l + S)/2) * T_b \\ + [(j_{sf} * ((R * (B/r)) * (S * (B/s))))) * (T_d + T_{io}) + (j_B * T_{bt})] \quad (123)$$

*5.1.3.6 Index. Case f.* This case provides indexes for both relations, providing an optimized environment for executing the join with index processing. The join processing with the indexes compares the indexes and retrieves the tuples that satisfy the join conditions. This processing must retrieve the tuples from random blocks. This may cause numerous disk accesses depending upon the placement and number of tuples to be retrieved. The performance model is a simplified version of the performance model of previous cases. Storing the results produces the following performance model:

## Model J - 20

$$(R_l + S_l) * (T_d + T_{io}) + ((R_l + S_l)/2) * T_b \\ + [(j_{sf} * ((R * (B/r)) * (S * (B/s))))) * 2(T_d + T_{io}) + (j_B * (T_d + T_{io}))] \quad (124)$$

Sending the results is:

## Model J - 21

$$(R_l + S_l) * (T_d + T_{io}) + ((R_l + S_l)/2) * T_b + \\ [(j_{sf} * ((R * (B/r)) * (S * (B/s))))) * 2(T_d + T_{io}) + (j_B * T_{bt})] \quad (125)$$

*5.1.3.7 Index. Case g.* This case provides one indexed-ordered relation and one unindexed-unordered relation. This provides several possible methods to complete the join. The first method would be to sort the unordered relation and then perform the join. However, this case is covered under the sort-merge cases. The other methods require an index to be constructed for the unindexed relation.

The first index method constructs the index for the unindexed relation and then uses the indexes for both relations to complete the join. This processing does not require the random accesses for the retrieval of the ordered relation because ordering and indexing provides a clustering index. This means that at most the entire relation must be read once to complete the retrieve of the necessary tuples. Therefore, the performance of this situation depends upon the selectivity factor. The fewer tuples that satisfy the join condition, the better the performance. The performance model reflecting this is:

#### **Model J - 22**

*Build index for S*

$$+(R_l + S_l) * (T_d + T_{io}) + ((R_l + S_l)/2) * T_b \\ + [(j_{sf} * ((R * (B/r)) * (S * (B/s))))) * 2(T_d + T_{io}) + (j_B * (T_d + T_{io}))] \quad (126)$$

Sending the results is:

#### **Model J - 23**

*Build index for S*

$$+(R_l + S_l) * (T_d + T_{io}) + ((R_l + S_l)/2) * T_b + \\ [(j_{sf} * ((R * (B/r)) * (S * (B/s))))) * 2(T_d + T_{io}) + (j_B * T_{bt})] \quad (127)$$

The other method to execute the join in this case is to use the ordered relation to be compared with the index of the other relation. This is the same processing as performed in case b. The performance model for this situation is:

## Model J - 24

*Build index for S*

$$\begin{aligned} & +(R + S_l) * (T_d + T_{io}) + ((R + S_l)/2) * T_b \\ & + [(j_{sf} * ((R * (B/r)) * (S * (B/s))))) * (T_d + T_{io}) + (j_B * (T_d + T_{io}))] \end{aligned} \quad (128)$$

Sending the results is:

## Model J - 25

*Build index for S*

$$\begin{aligned} & +(R + S_l) * (T_d + T_{io}) + ((R + S_l)/2) * T_b + \\ & [(j_{sf} * ((R * (B/r)) * (S * (B/s))))) * (T_d + T_{io}) + (j_B * T_{bt})] \end{aligned} \quad (129)$$

Neither of the methods provides the best performance time for all cases. The first method provides best performance when the results are small because it may not retrieve the entire ordered-indexed relation, only a few blocks. However, the second method provides better performance parameters when the results require all the blocks of the indexed-ordered relation to accesses because it eliminates the accesses and processing of the index blocks of the ordered relation. Which reduces the performance time. Therefore, the situation determines which performance model provides the best performance capability.

**5.1.3.8 Index. Case h.** This case like case c provides two ordered relations. Therefore, any method that requires an index to be constructed can not improve upon the merge processing of the sort-merge type join execution. However, this case provides the index for one of the relations. This provides the opportunity for executing the join using the index that may provide better performance than the sort-merge processing. This method would compare the index of the indexed relation with the ordered relation. This is the same method used in various previous cases. The performance model for this is:

### Model J - 26

$$(R_l + S) * (T_d + T_{io}) + ((R_l + S)/2) * T_b + [(j_{sf} * ((R * (B/r)) * (S * (B/s))))) * (T_d + T_{io}) + (j_B * (T_d + T_{io}))] \quad (130)$$

Sending the results is:

### Model J - 27

$$(R_l + S) * (T_d + T_{io}) + ((R_l + S)/2) * T_b + [(j_{sf} * ((R * (B/r)) * (S * (B/s))))) * (T_d + T_{io}) + (j_B * T_{bt})] \quad (131)$$

*5.1.3.9 Index. Case i.* This case allows two methods to be used to execute the join operation. The first method simply uses the indices of the two relations for the processing of the join. One feature of this method is that the tuple retrieval of the ordered relation uses a clustering index. This means that at most each block of the relation will be retrieved only once, not several times as the case may be with the non-clustering index (unordered relation). Therefore, the performance model for this method is:

### Model J - 28

$$(R_l + S_l) * (T_d + T_{io}) + ((R_l + S_l)/2) * T_b + \\ + \min \left| \begin{array}{c} [(j_{sf} * ((R * (B/r)) * (S * (B/s))))) * (T_d + T_{io})] \\ \text{or} \\ R * (T_d + T_{io}) \end{array} \right| \\ + \min \left| \begin{array}{c} [(j_{sf} * ((R * (B/r)) * (S * (B/s))))) * (T_d + T_{io})] \\ \text{or} \\ S * (T_d + T_{io}) \end{array} \right| + (j_B * (T_d + T_{io})) \quad (132)$$

Sending the results is:

### Model J - 29

$$\begin{aligned}
 & (R_l + S_l) * (T_d + T_{io}) + ((R_l + S_l)/2) * T_b + \\
 & + \min \left| \begin{array}{c} [(j_{sf} * ((R * (B/r)) * (S * (B/s))))] * (T_d + T_{io}) \\ \text{or} \\ R * (T_d + T_{io}) \end{array} \right| \\
 & + \min \left| \begin{array}{c} [(j_{sf} * ((R * (B/r)) * (S * (B/s))))] * (T_d + T_{io}) \\ \text{or} \\ S * (T_d + T_{io}) \end{array} \right| + (j_B * T_{bt}) \quad (133)
 \end{aligned}$$

The second method compares the ordered relation with the index of the unordered relation. The comparison of these two methods was previously discussed for case g. The performance model for this method is:

### Model J - 30

$$\begin{aligned}
 & (R + S_l) * (T_d + T_{io}) + ((R + S_l)/2) * T_b \\
 & + [(j_{sf} * ((R * (B/r)) * (S * (B/s))))] * (T_d + T_{io}) + (j_B * (T_d + T_{io})) \quad (134)
 \end{aligned}$$

Sending the results is:

### Model J - 31

$$\begin{aligned}
 & (R + S_l) * (T_d + T_{io}) + ((R + S_l)/2) * T_b + \\
 & [(j_{sf} * ((R * (B/r)) * (S * (B/s))))] * (T_d + T_{io}) + (j_B * T_{bt}) \quad (135)
 \end{aligned}$$

*5.1.3.10 Index. Case j.* This case provides indices and sequential ordering for both relations. Therefore, one of the methods of performing the join is merging the two relations. However, this method is part of the sort-merge type execution and is discussed under that heading. The methods of performing the join using the indices are: compare the indices and retrieve the necessary tuples or compare the index of one of the relations with the index of the other relation.

The advantage of comparing the indices of the two relations is that the index may require only a fraction of the blocks as the entire relation. Compounding this with the selection of only a small number of tuples that satisfy the join condition, it may have to access many fewer blocks to complete the join. Also, both of the relations have clustering indices. This means that at most the retrieval of tuples that satisfy the join condition will only read each block of the relation once. The performance model for this:

#### **Model J - 32**

$$(R_l + S_l) * (T_d + T_{io}) + ((R_l + S_l)/2) * T_b \\ + [(j_{sf} * ((R * (B/r)) * (S * (B/s))))] * 2(T_d + T_{io}) + (j_B * (T_d + T_{io})) \quad (136)$$

Sending the results is:

#### **Model J - 33**

$$(R_l + S_l) * (T_d + T_{io}) + ((R_l + S_l)/2) * T_b + \\ [(j_{sf} * ((R * (B/r)) * (S * (B/s))))] * 2(T_d + T_{io}) + (j_B * T_{bt}) \quad (137)$$

The second method would compare one of the entire relations with the index of the other relation. This method does not seem to provide a better performance capability except in the case where one of the relations is very small (i.e., the result of a select that selects only one or two tuples) and a very large relation. For this case the processing of the index of the small relation would only increase the number of blocks to be accessed. Therefore, this method would always compare the entire small relation with the index of the larger relation. The performance model for this is:

#### **Model J - 34**

$$(R + S_l) * (T_d + T_{io}) + ((R + S_l)/2) * T_b \\ + [(j_{sf} * ((R * (B/r)) * (S * (B/s))))] * (T_d + T_{io}) + (j_B * (T_d + T_{io})) \quad (138)$$

Sending the results is:

### Model J - 35

$$(R + S_l) * (T_d + T_{io}) + ((R + S_l)/2) * T_b + \\ [(j_{sf} * ((R * (B/r)) * (S * (B/s))))) * (T_d + T_{io}) + (j_B * T_{bt})] \quad (139)$$

## 5.2 Case 2. Join - Single Processor-Multiple Disks

The single processor-multiple disk environment does not allow the opportunity of using parallel processing but does provide for overlapping data access. Therefore, the following performance models are based upon the models for the single processor-single disk environment with overlapped disk I/O.

5.2.1 Nested-Loop. The nested-loop algorithm with multiple disk-single processor provides a small improvement over the nested-loop algorithm with a single processor-single disk by reducing the number of disk accesses necessary. It can be assumed that one disk is retrieving a portion of the input while another disk is seeking the proper location. The resulting performance equation reflects the reduced number of disk seeks but not all disk accesses are eliminated. This is caused by the fact that as much of the processor's memory as possible is dedicated to completing the join operation. Ullman has shown the effect of optimizing by the join by using more memory and by the selection of the control relation [79]. The equation when placing the results upon disk is:

### Model J - 36

$$T_c + [T_d + (p_b * T_{io})] * (R/p_b) + (R * S * T_b) + [(S * (R/p_b)) * T_{io}] + [j_B * (T_{io} + T_i)] \quad (140)$$

When sending the results to the user:

## Model J - 37

$$\begin{aligned}
 & T_c + [T_d + (p_b * T_{io})] * (R/p_b) \\
 & + (R * S * T_b) + [(S * (R/p_b)) * T_{io}] + [j_B * T_{bt}]
 \end{aligned} \tag{141}$$

### 5.2.2 Sort-Merge.

**5.2.2.1 Sort-Merge. Cases a, d, f.** These cases provide relations that are unordered. The processing then requires that both relations be sorted and then merged. Therefore, there are three phases to the processing: sort relation R, sort relation S, and merge the two relations. The first phase is to sort the unordered relation. Therefore, the following performance model is provided for the sorting of a relation with X blocks. The equation for the SortSm(X) is:

## Model J - 38

$$\begin{aligned}
 & T_d + (p_b * T_{io}) + T_{sort}(p_b, tb) + T_d + (p_b * T_{io}) \\
 & + \max \left| \begin{array}{c} ((X/p_b) - 1) * T_{sort}(p_b, tb) \\ \text{or} \\ ((X - p_b) * 2T_{io}) + (((X/p_b) - 1) * T_d) \end{array} \right| \\
 & + ([\log(X/p_b)] - 1) * (2T_d + 2T_{io}) + \max \left| \begin{array}{c} ((X/2) - 1) * T_b \\ \text{or} \\ (X - 2) * (T_d + 2T_{io}) \end{array} \right| + T_b + T_d + 2T_{io}
 \end{aligned} \tag{142}$$

This sort algorithm sorts all the blocks of data that can be contained in the memory of the processor. Then the sorted segments are merged to complete the sort of the relation. The merge is completed by merging two segments at a time. This provides larger and larger segments until all of the sorted segments are merged into the complete sorted relation.

The merge processing does allow overlapping of the disk I/O. The result of this is to reduce the number of disk accesses that would delay the processing of the merge.

The final segment of completing the sort merge algorithm is to merge the sorted relations. The performance model for the merge of the sorted relations storing the results on disk, MergeSMD(R,S), is:

#### Model J - 39

$$T_d + 2T_{io} + \max \left| \begin{array}{c} (((R + S)/2) - 1) * T_b \\ \text{or} \\ (((R + S) - 2) * T_{io})/d \\ + [j_B - 2] * T_{io} \end{array} \right| + T_b + T_d + 2T_{io} \quad (143)$$

If the results of the merge, MergeSMB(R;S), are sent directly to the user the following model is used:

#### Model J - 40

$$2T_d + 2T_{io} + \max \left| \begin{array}{c} (((R + S)/2) - 1) * T_b \\ + [j_B - 2] * T_{bt} \\ \text{or} \\ (((R + S) - 2) * T_{io})/d \end{array} \right| + T_b + 2T_{bt} \quad (144)$$

*5.2.2.2 Sort-Merge. Cases b, e, g, i.* All of these cases provide one relation that is already ordered using the attribute(s) necessary for evaluation of the join condition. This allows the sort-merge processing to sort the remaining relation and then execute the merge. Obviously, this will provide better performance than the previous cases. The models presented here represent relation R as the relation that is not sorted. But, this is an arbitrary identifier since R can be replaced by any value. Therefore, if S was the relation that was sorted the values of R and S would be interchanged for the computation of the execution time.

The sort-merge with one relation already sorted has two phases: sort the unordered relation and merge the sorted relations to complete the join. The performance equation is stated in terms of the separate performance components developed in the previous section. Thus, the performance model is:

**Model J - 41**

$$\begin{aligned} & SortSM(S) \\ & + MergeSMD(R,S) \end{aligned}$$

If the results are sent directly to the user, the following performance model is used:

**Model J - 42**

$$\begin{aligned} & SortSM(S) \\ & + MergeSMB(R,S) \end{aligned}$$

*5.2.2.3 Sort-Merge. Cases c, h, j.* These cases all present both relations already sorted on the attribute(s) needed for the join processing. This means the join only needs to merge the two relations to complete the join. The model for the merge, MergeSMD(R,S), holds for this case and MergeSMB(R,S) is the performance model when the results are sent directly to the backend.

*5.2.3 Indexing.* The first component of using the indices to perform the join operation is building the index when it does not exist. In the previous section, three methods of building the index were discussed. But, the first method of inserting individual tuples in no order creates a very large amount of disk I/O to find the correct place in the index each time. In fact as the number of tuples grows, the corresponding number of disk accesses increase by a factor of three or four or more depending upon the levels of the index. Also, the disk accesses are not overlapped with the processing time causing a significant amount of time to build the index.

Therefore, the first method of inserting each tuple individually to build the index will not be considered.

The other methods of building an index both involve sorting the relation or portions of the relation. The first method sorts the entire contents of the processor memory and then inserts this ordered segment. This method is very dependent upon the size of the processor memory and as the size of the memory approaches the size of the relation, it becomes the same model as the second index building method of sorting the entire relation first and then sequentially building the index. Also, both methods performance time is influenced by the size of the attribute(s) since the amount of data to be sorted in the attribute(s) and a TID. The performance equations for the build index equations, BuildSM(R), are:

### Model J - 43

$$((R/p_b) * T_d) + (R * T_{io}) + (R * (p_b/b) * T_s) + (R/p_b * T_{sort}) + (\text{total accesses}) \quad (145)$$

where total accesses is:

using  $l_b = ((i - 1) * tb * (p_b/z))$   
 $l_{b\_next} = (i * tb * (p_b/z))$   
 $tb = \text{number of tuples per block } (B/r)$   
 $z = \text{number of index entries per block } (B/(v + in))$

$$\sum_{j=1}^{R/p_b} \left\{ \begin{array}{l} l_b > (tb * p_b) \\ \text{or} \\ l_b \leq (tb * p_b) \end{array} \right\} \left\{ \begin{array}{l} +((tb * p_b) * (T_{ind} + T_{io} + T_d)) \\ +[l_{b\_next} - l_b] * T_{ind} \\ + \sum_{j=1}^{\log_2 l_{b\_next}} [((tb * p_b)/z) + ((1/(z/2) - 1) * tb * p_b)] * T_{ind} \\ +(l_{b\_next} * T_{ind}) + (l_b * (T_d + T_{io})) \\ + \sum_{j=1}^{\log_2 l_{b\_next}} [(i * tb * p_b) + ((1/(z/2) - 1) * l_{b\_next})] * T_{ind} \end{array} \right\} \quad (146)$$

and

where  $Rr = R * ((v + in)/r)$

### Model J - 44

$$\begin{aligned}
 & ((R/p_b) * T_d) + (R * T_{io}) + (R * (p_b/p) * T_s) + ((Rr/p_b) * T_d) \\
 & + (Rr * T_{io}) + (Rr * (p_b/b) * T_s) + ((R/p_b) * T_{sort}) + [(\log_{p_b-1}(Rr)) * (2 * Rr * (T_d + T_{io}))] \\
 & + ((Rr/2) * T_b) + (((R * tb)/z) * T_{ind}) + \sum_{i=0}^{\log_z(R * tb)/z} ((R * tb) * T_{ind}) \quad (147)
 \end{aligned}$$

The building of an index will be referred to as  $\text{Build}(X)$ , where X is the relation to be indexed. Next, after both relations are indexed the indices must be compared to identify the tuples that satisfy the join condition. To simplify the performance models, the comparing of the indexes and retrieval of the necessary tuples will be defined here as  $\text{IndexSM}(R,S)$ . The performance models for  $\text{IndexSM}(R,S)$  (depending upon the placement of the results) are:

### Model J - 45

$$\begin{aligned}
 & (R_l + S_l) * (T_d + T_{io}) + ((R_l + S_l)/2) * T_b \\
 & + [(j_{sf} * ((R * (B/r)) * (S * (B/s))))) * (T_d + 2T_{io}) + (j_B * (T_d + T_{io}))] \quad (148)
 \end{aligned}$$

Sending the results is:

### Model J - 46

$$\begin{aligned}
 & (R_l + S_l) * (T_d + T_{io}) + ((R_l + S_l)/2) * T_b + \\
 & [(j_{sf} * ((R * (B/r)) * (S * (B/s))))) * (T_d + T_{io}) + (j_B * T_{bt})] \quad (149)
 \end{aligned}$$

where  $R_l$  and  $S_l$  represent the number of leaf nodes in the index for R and S, respectively.  $R_l$  and  $S_l$  are calculated by:

(Number of Blocks \* Tuples per Block) \* (Attribute(s) to indexed + Address Size (TID)) / Block Size.

The other method of using an index to perform the join is to compare the index with an ordered relation. This operation OrInSm(R,S) where R is assumed to be unindexed but ordered, is modeled by:

#### **Model J - 47**

$$T_d + ((R + S_l) * T_{io}) + ((R + S_l)/2) * T_b \\ + [(j_{sf} * ((R * (B/r)) * (S * (B/s))))) * (T_d + T_{io}) + (j_B * T_{io})] \quad (150)$$

Sending the results is:

#### **Model J - 48**

$$T_d + ((R + S_l) * T_{io}) + ((R + S_l)/2) * T_b \\ + [(j_{sf} * ((R * (B/r)) * (S * (B/s))))) * (T_d + T_{io}) + (j_B * T_{bt})] \quad (151)$$

These models show a disk access time,  $T_d$ , for each tuple retrieval of the indexed relation since they are retrieved in random order. Notice that the multiple disk environment eliminates most of the other disk accesses after the initial access. However, the read time,  $T_{io}$ , is not eliminated because the processing of the memory segment is completed before the next segment is started. Therefore, the read time is not overlapped.

*5.2.3.1 Index. Case a.* Case A provide two relations that have no sequence and no index provided. This requires that an index be constructed for both relations before the index processing can begin. Therefore, the performance model is:

#### **Model J - 49**

$$BuildSM(R) \\ + BuildSM(S) \\ + IndexSm(R,S)$$

**5.2.3.2 Index. Case b.** This case provides one sorted relation and the other in random order. The processing using indices here has two options: build indices for both and process as in the previous case (except the ordered relation would require fewer disk accesses to retrieve results) or build an index for the random order relation and process the index with the sequential relation.

The ordered relation allows the building of the index to proceed more quickly because only the last segment of the indexing building is applied. However, the building of the index for the sorted relation requires reading the entire relation plus the time required to insert the attribute values into the index. After the indices are built, the join processing can begin. During the retrieval of tuples that satisfy the join condition, the clustering provided in the ordered relation, limits the accesses to that relation to one access per block. Whereas, the random sequence of the other relation, may require an access for each tuple retrieved. This means that building the index and accessing the tuples of the ordered relation requires the relation to be read twice plus the time required to build and process the index. If the ordered relation is used without building an index, the relation is read only once and compared to the index of the second relation. This requires the same processing for the join part of the processing but eliminates the time to build the index for the ordered relation thus, reducing the overall processing time to complete the join. Therefore, only the situation of building an index for the unordered relation and comparing this with the sorted relation will be modeled. This performance model is:

#### **Model J - 50**

$$BuildSM(R)$$

$$+ OrInSm(R,S)$$

**5.2.3.3 Index. Case c.** The situation where both relations are in sorted order does not provide a good environment for index processing of the join. If index processing of the join is to be accomplished, the first step is to build the indices. The

building of the indices requires the scanning of the relations, which requires reading all of the blocks of the relations. This, combined with processing time to build the index, is approximately equal to the time to perform a merge of the relation to execute the join. Therefore, the execution of the join with the indices would cause excessive time to be used. Thus, for the case of both relations being ordered and not indexed, the best method is the sort-merge approach and no index performance model will be shown.

*5.2.3.4 Index. Case d.* Case d provides one relation that is indexed and both relations are unordered. Therefore, the first step is to build the index for the unindexed relation. Then the processing of the join is completed by comparing the indexes. The performance model for this is:

#### **Model J - 51**

$$BuildSM(R)$$

$$+ IndexSM(R,S)$$

*5.2.3.5 Index. Case e.* This case provides one indexed relation and one ordered relation. This utilizes the OrInSm(R,S) algorithm to complete the join without any additional processing or building of indices. Therefore, the performance model is:

#### **Model J - 52**

$$OrInSM(R,S)$$

*5.2.3.6 Index. Case f.* This case provides indices for both relations, providing an optimized environment for executing the join with index processing. Therefore, the performance model is:

#### **Model J - 53**

$$IndexSM(R,S)$$

*5.2.3.7 Index. Case g.* This case provides one indexed-ordered relation and one unindexed-unordered relation. This provides several possible methods to complete the join. The first method would be to sort the unordered relation and complete the join by merging the ordered relations (this case is covered in a previous section on the sort-merge algorithm). Using the index methods, the index can be constructed for the unindexed relation and then either the indices compared or the ordered relation compared with index of the unordered relation. The performance model for comparing the indexes is:

**Model J - 54**

$$\begin{aligned} & \text{BuildSm}(S) \\ & + \text{IndexSM}(R,S) \end{aligned}$$

And when the ordered relation is compared to the index, the model is:

**Model J - 55**

$$\begin{aligned} & \text{BuildSm}(S) \\ & + \text{OrInSM}(R,S) \end{aligned}$$

*5.2.3.8 Index. Case h.* This case as case c provides two ordered relations. Therefore, any method that requires an index to be constructed can not improve upon the merge processing of the sort-merge type join execution. However, this case provides the index for one of the relations. Therefore, the performance model is:

**Model J - 56**

$$\text{OrInSM}(R,S)$$

**5.2.3.9 Index. Case i.** This case has two relation that have indices already constructed. Thus, the first method of completing the join is to compare the indices. The performance model for this is:

#### Model J - 57

$$IndexSM(R,S)$$

Since one relation is ordered, the tuple retrieval will at most read each tuple once. Therefore,

$$\begin{aligned} & [(j_{sf} * ((R * (B/r)) * (S * (B/s))))) * (T_d + 2T_{io}) \\ & \leq \\ & [(j_{sf} * ((R * (B/r)) * (S * (B/s))))) * (T_d + T_{io}) + (R * T_{io}) \end{aligned} \quad (152)$$

The second method compares the ordered relation with the index of the un-ordered relation. The model for this is:

#### Model J - 58

$$OrInSM(R,S)$$

**5.2.3.10 Index. Case j.** This case provides indices and sequential order for both relations. Therefore, one of the methods of performing the join is merging the two relations. However, this method is part of the sort-merge type execution and is discussed under that heading. The methods of performing the join using the indices are: compare the indices and retrieve the necessary tuples or compare the index of one of the relations with the index of the other relation.

The advantage of comparing the indices of the two relations is that the index may require only a fraction of the blocks as the entire relation. Compounding this with the selection of only a small number of tuples that satisfy the join condition, it may have to access many fewer blocks to complete the join. Also, both of the relations have clustering indexes. This means that at most the retrieval of tuples

that satisfy the join condition will only read each block of the relation once. The performance model for this:

#### Model J - 59

$$IndexSM(R,S)$$

Where

$$[(j_{sf} * ((R * (B/r)) * (S * (B/s))))) * (T_d + 2T_{io}) \leq (R + S) * T_{io} \quad (153)$$

The second method compares one of the entire relations with the index of the other relation. This method does not seem to provide a better performance capability except in the case where one of the relations is very small (i.e., the result of a select that selects only one or two tuples) and a very large relation. For this case the processing of the index of the small relation would only increase the number of blocks to be accessed. Therefore, this method would always compare the entire small relation with the index of the larger relation. The performance model for this is:

#### Model J - 60

$$OrInSM(R,S)$$

### 5.3 Case 3. Join - Multiple Processors-Single Disk

The multiple processor-single disk environment must assume that each processor can access the disk. Also, it is assumed that all processors can communicate with each of the other processors. Other considerations will be discussed in connection with the algorithm as it effects the processing.

*5.3.1 Nested-Loop.* Several assumptions can be made about the processing and control structures that may effect the processing of the nested-loop join algorithm. It must be remembered that the nested-loop algorithm executes the join by

comparing each tuple of one relation with each tuple of the other relation. This requires some processor to control the processing. If it is assumed that the processors are synchronized, then a block can be broadcast to each processor requiring these messages being passed to control the processing. If the blocks cannot be broadcast to a group of processors, each processor must pass a control message to a controller requiring numerous message traffic and random disk accesses to provide a block of data at different times. Therefore, the performance model assumes that the blocks of data can be broadcast to the processors. Using this the performance model is:

#### **Model J - 61**

$$T_c + T_m + [2T_d + ((p_b * p)/b) * T_s + ((p_b * p) * T_{io})] * (R/(p_b * p)) \\ + [(R/p) * S * T_b + (S * (R/(p_b * p))) * T_{io} + [j_B * (T_{io} + 2T_d)] \quad (154)$$

When sending the results to the user:

#### **Model J - 62**

$$T_c + T_m + [2T_d + ((p_b * p)/b) * T_s + ((p_b * p) * T_{io})] * (R/(p_b * p)) \\ + [(R/p) * S * T_b + (S * (R/(p_b * p))) * T_{io} + [j_B * T_{bt}] \quad (155)$$

**5.3.2 Sort-Merge.** The first element of the sort-merge processing is providing a sorted relation. Therefore, the performance equation for sorting a relation in the multiple processor-single disk environment, SortMS(x), is:

## Model J - 63

$$T_d + (p_b * T_{io}) + Tsort(p_b, tb) + T_d + (p_b * T_{io})$$

$$+ \max \left| \begin{array}{c} ((X/(p * p_b)) - 1) * Tsort(p_b, tb) \\ \text{or} \\ (((X/b) - 1) * T_s) + ((X - p_b) * 2T_{io}) + (((X/p_b) - 1) * 2T_d) \end{array} \right| + 2T_d + 2T_{io}$$

$$+ \max \left| \begin{array}{c} ((X/(p * p_b))/2) * \log(((X/(p * p_b)/2))) * T_b \\ \text{or} \\ (p * (((X/(p * p_b)/2) * \log(((X/(p * p_b)/2))) - 1)) * (2T_d + 2T_{io} + T_d + 2T_{io})) \end{array} \right|$$

$$+ ((T_d + 2T_{io}) * p) + T_b + 2T_d + 2T_{io} + \max \left| \begin{array}{c} (((X/(p * p_b)/2) * T_b) \\ \text{or} \\ ((X/2) - 1) * (2T_d + 2T_{io} + T_d + 2T_{io})) \end{array} \right|$$

$$+ ((T_d + 2T_{io}) * p) + T_b + 2T_d + 2T_{io} + \max \left| \begin{array}{c} (((X/(p * p_b)/2) * T_b) + (\log(p - 1) * T_{bt})) \\ \text{or} \\ ((X/2) - 1) * (2T_d + 2T_{io} + T_d + 2T_{io}) \end{array} \right|$$

$$+ T_d + 2T_{io} + T_b$$

(156)

The second part of the sort-merge processing of the join is the merging of the two sorted relations. The merge compares the individual tuples to determine if they satisfy the join conditions. The merge scans each entire relation once during its process cycle. However, since the sorted relations cannot be divided evenly because they may have different value distributions, the merge is considered to be a single processor task. Therefore, in the multiple processor environment several processors may be idle during this portion of the processing. It would seem that the proper

method would be to simultaneously sort the relations passing the results to a single processor. However, in the single disk environment the concurrent sorts would be contending for the same disk accesses. Thus, changing the sort equations from two separate equations to one equation that would have to delay each disk I/O because of the disk contention problem. Therefore, for this architecture the merge will not worry about idle processors. In the multiple disk-multiple processor environment the possibility of concurrent sorting of the two input relations will be explored.

The performance model for the merge in the multiple processor-single disk environment becomes the same model as the single processor-single disk model presented earlier. Therefore, the merge model, MergeMS(R,S), is:

#### **Model J - 64**

$$2T_d + 2T_{io} + \max \left| \begin{array}{l} (((R+S)/2) - 1) * T_b \\ \text{or} \\ ((R+S) - 2) * (T_d + T_{io}) \\ + [j_B - 2] * (T_d + T_{io}) \end{array} \right| + T_b + T_d + 2T_{io} \quad (157)$$

And when sending the results to the backend the model is:

#### **Model J - 65**

$$2T_d + 2T_{io} + \max \left| \begin{array}{l} (((R+S)/2) - 1) * T_b \\ + [j_B - 2] * T_{bt} \\ \text{or} \\ ((R+S) - 2) * (T_d + T_{io}) \end{array} \right| + T_b + 2T_{bt} \quad (158)$$

*5.3.2.1 Sort-Merge. Cases a, d, f.* The relations in these cases are both unordered. Therefore, the relations are first sorted and then merged to complete the join operation. The performance model for this is:

#### **Model J - 66**

$$SortMS(R)$$

$$\begin{aligned}
 & + \text{SortMS}(S) \\
 & + \text{MergeMS}(R,S)
 \end{aligned}$$

*5.3.2.2 Sort-Merge. Cases b, e, g, i.* The relations in these cases are one sorted relation and one unordered relation. Therefore, to complete the sort-merge, first the unordered relation is sorted and then the ordered relations are merged to complete the join. The performance model for this is (substituting R or S for X as appropriate):

**Model J - 67**

$$\begin{aligned}
 & \text{SortMS}(S) \\
 & + \text{MergeMS}(R,S)
 \end{aligned}$$

*5.3.2.3 Sort-Merge. Cases c, h, j.* These cases all present both relations already sorted on the attribute(s) needed for the join processing. This means the join only needs to merge the two relations to complete the join. Therefore, the model for join for these cases is MergeMS(R,S).

*5.3.3 Indexing.* The use of the index to complete the join to this point has assumed that each relation had one centralized index. Processing of the join, using the centralized indices, was one done by a single processor since that was all that was available. This environment provides more processors to be used in the processing but it is assumed that a single processor would process the join due to the inability of the disk to provide the support to more than one processor due to the random nature of retrievals from the disk. Therefore for the explanation and models for this case, refer to the single processor-single disk architecture indexing section.

#### *5.4 Case 4. Multiple Processors-Multiple Disks*

The multiple processor-multiple disk environment requires several assumptions about the element of the architecture, such as: processor communication capability,

disk-processor interconnection, data distribution, processor control, and others, to be made. These assumptions all may impact the performance time of the processing the query. In particular, the effect is especially apparent for the binary operators such as the join. The first element to be considered is the data distribution.

A relation may be distributed over several disks in several different ways. In a previous chapter, the different ways of partitioning a relation were discussed. However, partitioning a relation may be different from distributing the data contained in the relation because even a vertical fragment of a relation could be stored on two or more different storage devices. Therefore, the only consideration here, whether it is a vertical fragment or a horizontal fragment, is that each tuple is a complete entity for storage purposes. This just means that a complete tuple is stored in one location, no tuples may be split across storage devices (this considers each vertical fragment to contain tuples). Therefore, the following distributions will be explored:

1. Even distribution - this places an equal number of blocks of data on each disk.
2. Bucket distribution - this distribution hashes each tuple to a disk based upon some hash function applied to the key attribute of the tuple.
3. Ordered distribution - this distribution maintains the tuples in order over all the disks.

The first distribution, the even distribution, provides no sequencing by value. Its only consideration is to try to maintain an equal amount of blocks of the relation on each disk. Therefore, a round-robin algorithm determines the placement of additional tuples.

The second distribution groups the tuples by some hashing scheme. This produces groups or buckets of tuples that are evaluated to be equal using the hashing method. This method does not necessarily provide an even distribution of data across all the storage devices. Also, within the disk, the tuples may be maintained in random order or sorted in some sequential order based upon the ordering attribute.

The third distribution or the ordered distribution maintains a sequential ordering of the relation. This ordering is maintained within a disk and across all disks based upon some key attribute(s). An equal distribution of data across the disk may be maintained depending upon the reorganization techniques used during the insertion of new tuples.

The various data distribution schemes may influence the type of algorithm used to complete the join. Also, the interconnectability of processor and disk may affect the performance models. For the performance models that follow in this chapter it is assumed that each processor can access each disk or that the communication time to send data from a disk to a specific processor does not increase the processing time of an intermediate processor. Also, for this set of performance models the processors are modeled as being fully interconnected. Therefore, the performance discussion models the message or block transfer time as a constant. This time may not be a constant value in current multiple processor implementations but trying to model each different communication configuration becomes overwhelming.

Three basic type algorithms for executing the join will be discussed. First, the nest-loop algorithm will be modeled. Then, the sort-merge algorithm will be evaluated. And finally, the indexed and bucket sort type join algorithm will be explored.

*5.4.1 Nested-Loop.* The nested-loop algorithm compares each tuple of the first relation with each tuple of the second relation. This method may not be the most efficient algorithm for executing the equi or natural join. But, this method does allow any join condition, including the Cartesian product, and performs them in the same amount of time (not including time to store or send the results, which vary with each query). The nested loop algorithm model assumes that the R relation has fewer blocks than the S relation. It is also assumed that a disk can broadcast the same block to all the processors in  $T_{io}$  time. Using these assumptions the nested-loop

algorithm performance model is:

### Model J - 68

$$T_c + T_m + [T_d + (((p_b * p)/d)/b) * T_s + (((p_b * p)/d) * T_{io})] * (R/(p_b * p)) \\ + ((R/p) * S * T_b) + [(S * (R/(p_b * p))) * T_{io}] + [(j_B/d) * (T_{io} + T_d)] \quad (159)$$

When sending the results to the user:

### Model J - 69

$$T_c + T_m + [T_d + (((p_b * p)/d)/b) * T_s + (((p_b * p)/d) * T_{io})] * (R/(p_b * p)) \\ + ((R/p) * S * T_b) + [(S * (R/(p_b * p))) * T_{io}] + [(j_B/d) * T_{ot}] \quad (160)$$

Since the nested-loop algorithm physically compares all the tuples, there is no difference in processing time for any of the data distributions discussed. Therefore, the processing time is the same for all distributions (and for all join conditions).

*5.4.2 Sort-Merge.* The first question in using the sort-merge algorithm is how to execute the sort in the multiple processor environment. The sort algorithm could be concerned with such issues as distributing the sort to all the processors and having the processors perform the sort in parallel and how important to keep all processors busy or is it allowable to let some processors be idle. The sort algorithm used here is a parallel binary merge sort that sorts the initial blocks and then merges the original sorted segments into larger sorted segments until the last merge merges two segments to produce the single sorted relation [8]. The equation for SortMM(X) is:

## Model J - 70

$$\begin{aligned}
 & T_d + (p_b * T_{io}) + T_{sort}(p_b, tb) + \max \left| \begin{array}{l} ((X/(p * p_b)) - 1) * T_{sort}(p_b, tb) \\ \text{or} \\ (((X/d) - p_b)/b) * T_s \\ + (((((X/d) - p_b) + ((X/d) - p_b) * T_{io}) \\ + (((X/p_b)/d) - 1) * 2T_d) \end{array} \right| \\
 & + T_{sort}(p_b, tb) + T_d + (((p/d) * p_b) * T_{io}) + (((p/d) * p_b)/b) * T_s + 2T_d + 2T_{io} \\
 & + \max \left| \begin{array}{l} ((X/(p * p_b))/2) * \log(((X/(p * p_b))/2)) * T_b \\ \text{or} \\ [(p * (((X/(p * p_b))/2) * [\log((X/(p * p_b))/2)] - 1)/d)] * (2T_d + 2T_{io} + T_d + 2T_{io}) \end{array} \right| \\
 & + ((T_d + 2T_{io}) * (p/d)) + T_b + 2T_d + 2T_{io} \\
 & + \max \left| \begin{array}{l} ((X/(p * p_b))/2) * T_b \\ \text{or} \\ ((X/d)/2) * (2T_d + 2T_{io} + T_d + 2T_{io}) \end{array} \right| + ((T_d + 2T_{io}) * (p/d)) \\
 & + T_b + 2T_d + 2T_{io} + \max \left| \begin{array}{l} (((X/(p * p_b))/2) * T_b) + (\log(p - 1) * T_{bt}) \\ \text{or} \\ ((X/d)/2) * (2T_d + 2T_{io} + T_d + 2T_{io}) \end{array} \right| + T_d + 2T_{io} + T_i
 \end{aligned} \tag{161}$$

The other phase of the sort-merge join is the merge processing. Even though there are multiple disks and multiple processors, the merge is still best accomplished by a single processor because the sorted relations do not have points to divide the relations that insure that the dividing point is the same for both relations except scanning the relations. And, if the relations are to be scanned, then the join process-

ing may as well be accomplished. The number of disks involved in the merge depend upon the distribution of the results of the sorts. However, assuming that each processor can access each disk, this does not impact the merge model. Therefore, the MergeMM(R,S) model is:

### Model J - 71

$$T_d + 2T_{io} + \max \left| \begin{array}{l} (((R+S)/2) - 1) * T_b \\ \text{or} \\ (((R+S) - 2) * T_{io}) \\ + ((j_B - 2) * T_{io}) \end{array} \right| + T_b + 2T_{io} \quad (162)$$

and

### Model J - 72

$$T_d + 2T_{io} + \max \left| \begin{array}{l} (((R+S)/2) - 1) * T_b \\ + ((j_B - 2) * T_{bt}) \\ \text{or} \\ (((R+S) - 2) * T_{io}) \end{array} \right| + T_b + 2T_{bt} \quad (163)$$

storing the results and sending the results to a backend, respectively.

*5.4.2.1 Sort-Merge. Cases a, d, f.* This model does not reflect the possibility of pipelining the results of the sorts directly to the merge which would eliminate the writing of the sorted results to disk and the reading of the sorted relation from disk for processing by the merge.

The relations in these cases are both unordered. Therefore, the relations are first sorted and then merged to complete the join operation. The performance model for this is:

### Model J - 73

$$SortMS(R)$$

$$\begin{aligned}
 & + SortMS(S) \\
 & + MergeMS(R,S)
 \end{aligned}$$

*5.4.2.2 Sort-Merge. Cases b, e, g, i.* The relations in these cases are one sorted relation and one unordered relation. Therefore, to complete the sort-merge, first the unordered relation is sorted and then the ordered relations are merged to complete the join. The performance model for this is (substituting R or S for X as appropriate):

#### Model J - 74

$$\begin{aligned}
 & SortMS(X) \\
 & + MergeMS(R,S)
 \end{aligned}$$

*5.4.2.3 Sort-Merge. Cases c, h, j.* These cases all present both relations already sorted on the attribute(s) needed for the join processing. This means the join only needs to merge the two relations to complete the join. Therefore, the model for join for these cases is MergeMS(R.S).

The models presented do not reflect the possibility of pipelining the results of the sort(s) directly to the merge which would eliminate the writing of the sorted results to disk and the reading of the sorted relation from disk for processing by the merge. The processing for this would use the sending the results to the backend model for the sort and eliminate appropriate the disk accesses for retrieving the sorted data from the merge. Therefore, if both relations need to be sorted, each sort would be allotted a portion of the processors to accomplish the sort. This would require more time to accomplish each individual sort, but at the final phases of the sort, processors are released which allows a processor to be free to perform the merge. Since the final steps of the sorts are merges and the final step of the sort-merge join is a join, the operations will perform without wait in a pipeline situation. Thus, the merge will be completed in  $T_b + T_{bt}$  time beyond the time to complete the sorts.

subsectionIndexing.

The indexed case because of its centralized nature does not provide a significant opportunity for parallel processing. Research is being done on parallel processing of indices [63], but this research requires unique hardware to perform the parallel processing of the index. However, in this environment the largest potential may arise from the parallel retrieval of the tuples that were identified by the index processing. This does not imply that the building of an index can be done in parallel. Therefore, the index case has very limited potential when both relations are unindexed.

The first step using indices to perform a join is to provide the indices. If an index has to be constructed, the relation can be constructed using the methods described for previous architectures. However, the index is centralized which constrains the number of processors that can concurrently be updating the index. To utilize more parallel processing, the concept of projecting out the indexing attribute(s) and sorting these results to build the index. This method allows several processors to operate concurrently to perform the projection operation (adding the TID). This in a pipeline with a processor constructing the index provides an index building algorithm that utilizes some parallel processing. The performance model for this ( $\text{BuildInMM}(X)$ ) is:

Model J - 75

$$\begin{aligned}
 & T_c + T_m + T_d + T_{io} + \max \left| \begin{array}{l} ((R/p) - 1) * T_{sc} \\ + [((p_f/(p * p_b)) - 1) * T_{sort}] \\ \text{or} \\ (((R/d) - 1)/b) * T_s \\ + [(((R/d) - 1) + ((p_f/d) - 1) * T_{io}] \\ + [([(p_f/p_b)/d] - 1) * 2T_d] \end{array} \right| \\
 & + T_{sc} + T_{sort} + T_d + ((p/d) * T_{io}) + ((p/d)/b * T_s) \\
 & + 2T_d + 2T_{io} + \max \left| \begin{array}{l} (p_f/2p) \log(p_f/2p) * T_b \\ \text{or} \\ (p * [p_f/2p \log(p_f/2p)] - 1)/d * [(2T_d + 2T_{io}) + (T_d + 2T_{io})] \end{array} \right| \\
 & +(T_d + 2T_{io}) * (p/d) + T_b + 2T_d + 2T_{io} + \max \left| \begin{array}{l} (p_f/2p) * T_b \\ \text{or} \\ (p_f/2)/d * [2T_d + 2T_{io}] \\ + (T_d + 2T_{io}) \end{array} \right| \\
 & +(T_d + 2T_{io}) * (p/d) + T_b + 2T_d + 2T_{io} \\
 & + \max \left| \begin{array}{l} \max \left| \begin{array}{l} [((p_f/2) - 1) * T_b] \\ +(p_f - 2) * T_{bt} \\ \text{or} \\ ((p_f/2)/d) * (2T_d + 2T_{io}) \end{array} \right| + T_m + 2T_{bt} \\ \text{or} \\ T_b + T_{bt} + p_f * (T_{sc} + T_{io}) + \sum_{i=0}^{\log_2^{(R*tb)/z}} ((R * tb) * T_{ind}) \end{array} \right| \\
 & + 2T_{sc} + 2T_{io} + (L_I * T_{ind})
 \end{aligned}$$

(164)

where  $p_f = ((v + in) * ((R * B)/r))/B)$

This equation uses the blocks from the sort of the projection of the index value to build the index. This allows the overlapping of the final merge of the sort and the building of the index.

The next phase of the join using indices is the actual processing of the join. This requires the leaf nodes of the index to be compared and when matching values occur, the TIDs are retained to retrieve the tuples that form the results of the join. Since the leaf nodes of the index provide a sequential order, this is just a merge of the indices. And as discussed earlier, only one processor can effectively be used to perform a merge. This means that other processors are idle and could be used to retrieve the necessary tuples. Therefore, the retrieving of tuple may be overlapped with the merge processing. The processing of the indices and retrieving the necessary tuples, InJoin(R,S), is:

### Model J - 76

$$\begin{aligned}
 & T_d + T_{io} + T_b + \max \left| \begin{array}{l} (((R_l + S_l)/2) - 1) * T_s \\ \text{or} \\ \max \left| \begin{array}{l} (R_l * T_{io}) + ((R_l/b) * T_s) \\ \text{or} \\ (S_l * T_{io}) + ((S_l/b) * T_s) \end{array} \right. \right. \\ & \quad \text{or} \\ & \quad \left. \left. \left[ (j_{sf} * ((R * (B/r)) * (S * (B/s)))) \right] / (d - 2) \right] * (T_d + T_{io}) \right. \\ & \quad \left. \left. - (j_B * T_{bt}) / (p - 2) \right. \right. \end{array} \right. \quad (165)
 \end{aligned}$$

These two equations provide the basis for executing the join using the indices. There is no need to describe all the separate cases as each has been described before and all the execution requires is to plug in the proper modules discussed immedi-

ately above. Therefore, if the case did not provide an index for either relation, the performance model would be:

#### Model J - 77

$$BuildInMM(R)$$
$$+ BuildInMM(S)$$
$$+ InJoin(R,S)$$

This model uses the index for processing but an alternative would stop the building of the index and just use the sorted list of index values and TIDs to be matched. This method does not provide an index that could possibly be retained for later use. But, it reduces the processing time by eliminating the building an index that cannot be used further in processing this query. The processing of the indices for the join simply retrieves the same sequential list of values and TIDs. The actual performance model for this would use the project model when the results are stored on disk for the time to build the sorted value list and then the same join model as currently employed. These methods still depend upon the speed of retrieval and number of random tuples that need to be retrieved for the results of the join. If the number is large, this will be a large time requirement. But if the number of tuples is small, the time may be less than other methods of performing the join.

*5.4.3 Bucket Join or Hash Join.* The purpose of the join is to compare each tuple of the two relations to determine if they meet the join conditions. Since only the equi-join is considered, we use some “tricks” to reduce the number of tuples that actually have to be compared. The nested-loop algorithm literally compares each of the tuples. The sort-merge algorithm uses one of the “tricks” by ordering the tuples. This insures that the match only has to scan the lists once due to this ordering. The index processed join also uses the concept of sequential ordering of the join attributes for processing. But, it seems that there should be some other method to be able to

group tuples so that not all the tuples would have to be physically compared. That is what the bucket join is.

The bucket or hash join scans each relation, using the join attribute(s) as input to a hashing algorithm. The hashing algorithm produces a hash value upon which the tuples are split into groups. The same hashing algorithm is used upon both relations. This produces groupings of tuples from each relation to be physically compared. We are sure the groupings provide the correct tuples for comparison because the same hash function is applied to the same attribute(s) (defined over the same domain). If different hash functions were used or the hashing occurred over different attributes, there would be no assurance that the proper tuples were being grouped for comparison.

The groupings of tuples or buckets provide the name of this method. The actual implementation of this method would scan the relations applying the hash function. The tuples of the various groups would be sent to a processor designated for that particular value. Therefore, a processor would receive tuples from both relations. Then the processor would join the tuples it received. This could be accomplished by any of the join algorithms previously described. Since no ordering or index is retained through the hashing operation, the nested-loop join will be used here to join the individual buckets.

The critical factor of the bucket join is having a hash function that evenly distributes the tuples for processing. Assuming an even distribution of the buckets, the other significant factor is inter-processor communication. The performance model, that follows, assumes full processor interconnectivity. The model does assume that the data received by a processor during the hashing or splitting into bucket phase is larger than can be contained in the processor memory causing the data to be saved on disk. Also, there can be some variation of the number of processors used to process the relations and the number of processors used for the second phase of the join.

The performance model assumes that all processors are used to process the relations.

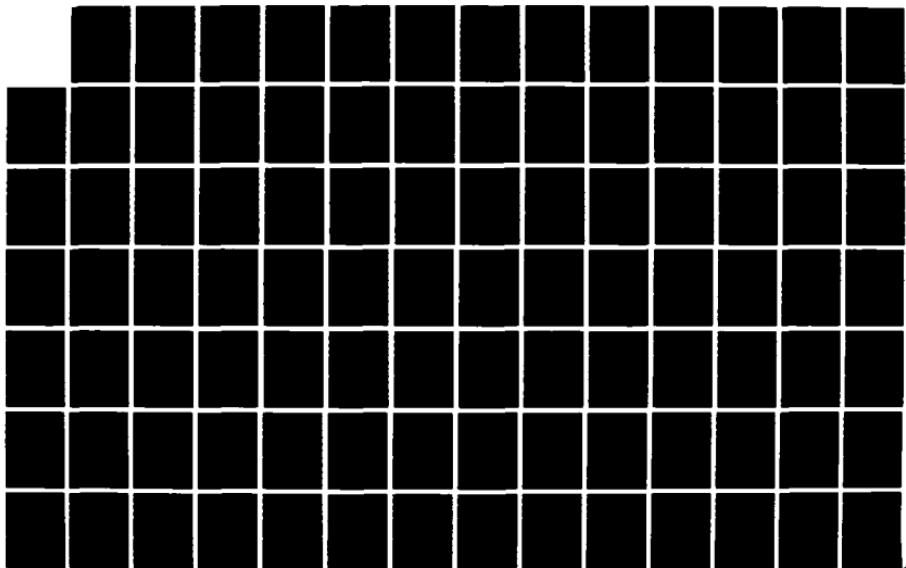
RD-R189 844

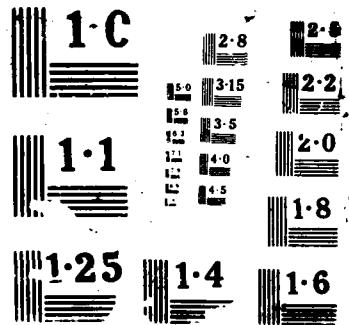
A METHODOLOGY BASED ON ANALYTICAL MODELING FOR THE  
DESIGN OF PARALLEL AND. (U) AIR FORCE INST OF TECH  
WRIGHT-PATTERSON AFB OH SCHOOL OF ENGI.. T G KEARNS  
UNCLASSIFIED DEC 87 AFIT/DS/ENG/87-1

3/4

F/G 12/6

NL





Also, additional time is included for receiving the blocks and storing them during the hashing phase. Using these assumptions, the performance model when sending the results to disk is:

### Model J - 78

$$\begin{aligned}
 & T_c + T_m + T_d + T_{io} + \max \left| \begin{array}{c} (T_{sc} * (R/p)) \\ +(2 * [((R/p) + 1) * (p - 1)] * T_{bt}) \\ \text{or} \\ 2 * ((R/d) * T_{io}) + (((R/d)/b) * T_s) \\ +((R/p) + 1) * (T_d + T_{io}) \end{array} \right| \\
 \\
 & + T_d + T_{io} + \max \left| \begin{array}{c} (T_{sc} * (S/p)) \\ +(2 * [((S/p) + 1) * (p - 1)] * T_{bt}) \\ \text{or} \\ 2 * ((S/d) * T_{io}) + (((S/d)/b) * T_s) \\ +((S/p) + 1) * (T_d + T_{io}) \end{array} \right| \quad (166)
 \end{aligned}$$

$$\begin{aligned}
 & + \{[T_d + (p_b * T_{io})] * (((R/p) + 1)/p_b)\} + (((R/p) + 1) * ((S/p) + 1) * T_b) \\
 & + [(((S/p) + 1) * (((R/p) + 1)/p_b)) * T_{io}] + [(j_B/p) * (T_{io} + T_d)]
 \end{aligned}$$

When sending the results to the user:

## Model J - 79

$$T_c + T_m + T_d + T_{io} + \max \left| \begin{array}{l} (T_{sc} * (R/p)) \\ +(2 * [((R/p) + 1) * (p - 1)] * T_{bt}) \\ \text{or} \\ 2 * ((R/d) * T_{io}) + (((R/d)/b) * T_s) \\ +((R/p) + 1) * (T_d + T_{io}) \end{array} \right|$$

$$+T_d + T_{io} + \max \left| \begin{array}{l} (T_{sc} * (S/p)) \\ +(2 * [((S/p) + 1) * (p - 1)] * T_{bt}) \\ \text{or} \\ 2 * ((S/d) * T_{io}) + (((S/d)/b) * T_s) \\ +((S/p) + 1) * (T_d + T_{io}) \end{array} \right| \quad (167)$$

$$+\{[T_d + (p_b * T_{io})] * (((R/p) + 1)/p_b)\} + (((R/p) + 1) * ((S/p) + 1) * T_b)$$

$$+[((S/p) + 1) * (((R/p) + 1)/p_b)) * T_{io}] + [(j_B/p) * T_{bt}]$$

### 5.5 Summary

The models presented are for the execution of the equi-join operations. The various algorithms perform the join operation by either some form of grouping the relations to reduce the processing or by physically comparing each tuple of one relation with each tuple of the other relation. Due to the varying methods of grouping the relations for processing, direct comparison of the analytical models is difficult because each may provide the best performance for some given combination of size and performance parameters. However, for the join operation for other than equality conditions only one algorithms is considered.

The other than equality condition normally invokes the same processing as the product operator. This involves comparing each tuple of the one relation with each

tuple of the second relation. The only algorithms considered here that provides this is the nested-loop algorithm. The nested-loop algorithm also is the only method of performing a product operation where each tuple of one relation is combined with each tuple of the second relation. Therefore, the models presented here for the nested-loop algorithm also provide the performance models for the product operation.

## *VI. Update Performance Effects*

Updating a database has not often been considered in the performance considerations of a database machine. However, in this electronic age many applications cannot depend upon information that was updated 16 hours earlier during the last batch update. Therefore, this chapter reflects the performance effects of providing immediate updates with the various data models.

There are three update operations: inserting new tuples, deleting a tuple, and modifying a tuple. These operations also have two parts integrity checking and the actual modification of the relation. By definition, there is no duplication of tuple in a relation [17]. The integrity checking assumed here fulfills the check for no duplicates being introduced by any update action. However, no referential integrity [17] checking is included. Referential checking involves performing selections on other relations to insure the references do exist. If referential integrity checking is to be modeled, it consists of select(s) plus the modeling of the update action (this assumes that the relations are in at least third normal form).

The first update operation is the inserting of a new tuple. The next operation discussed will be the deletion. And finally, the modification will be explored.

### *6.1 Inserting a new tuple.*

The insertion of a new tuple has two distinct phases—integrity checking and updating. The first phase of the insert is to insure the new tuple does not duplicate an existing tuple. In some special cases, the introduction of a duplicate cannot occur or the existence of a duplicate tuple does not matter. However, it will be assumed here that it is necessary to insure the integrity of the database by checking for duplicates. Further integrity checking for functional dependency type considerations will not be considered because these could be considered as separate selections from other relations.

The integrity check for duplicates is a select with a selection value given for the attribute(s) that form the key for the relation. If the select finds a matching tuple, then the insertion operation is terminated. Therefore, the select case where few tuples are expected matches the situation of checking for a duplicate tuple. Next, the individual architectures and data structures will be used to fully develop the performance models for the insertion operation.

#### 6.1.1 Single Processor - Single Disk Insertion.

6.1.1.1 *Unordered - Unindexed Insertion.* The unorganized data structure is the easiest and hardest data structure to accomplish an insertion. The hardest part is determining if the tuple duplicates another tuple. This requires reading and scanning the entire relation to insure that no duplicate exists. However, the easiest part is the actual insertion. The insertion simply adds the new tuple to the last block of the relation (if there is room, otherwise a new block is written) and writes that block back on disk. Thus, the performance models inserting a tuple into relation R is:

##### Model U - 1

$$T_c + T_d + T_{io} + \max \left| \begin{array}{c} (R - 1) * T_{sc} \\ \text{or} \\ (((R/b) - 1) * T_s) + ((R - 1) * T_{io}) \end{array} \right| + T_{sc} + T_{io} \quad (168)$$

6.1.1.2 *Unordered - Indexed Insertion.* The indexed data structure allows the index to be searched to determine if the tuple to be inserted is a duplicate. However, this assumes that the index is for the key of the relation since in a relational database the key of the relation cannot be duplicated. If the index was only for one attribute which was only a portion of the key, then the index would identify the tuple(s) to be retrieved to be examined to see if the tuple to be inserted is a duplicate.

Once the tuple is determined to be a valid tuple for insertion, two things remain to be done. First, a block with space for the tuple is retrieved or a new block created, the tuple is added to the block, and the block is written to disk. Then the index must be updated by adding the new value and TID to the index. The updating of the index should only require adding the value and TID to the last index block retrieved. Thus, this would just require the tuple to be added to the block and the block rewritten. Also, in some instances the block must be split which causes further updates of higher level blocks.

The performance model for the indexed case must approximate the additional processing necessary when the index block is caused to be split into two blocks. Using the approximation of the average number of splits,  $1/(\lceil z/2 \rceil - 1)$ , the performance model is:

### Model U - 2

$$T_c + ((L_I + 1) * T_{ind}) + T_{io} + T_d + T_{io} + T_{sc} + T_{io} + [(1/(\lceil z/2 \rceil - 1)) * 2T_{ind}] \quad (169)$$

*6.1.1.3 Ordered - Unindexed Insertion.* The ordered case presents a unique problem with inserting a new tuple. The first step checks the relation for a duplicate tuple. Since the relation is stored in sequential order, only a portion of the relation must be scanned to determine if a duplicate tuple exists. The expected value of finding the correct location in the relation for where the new tuple should be inserted and where a duplicate tuple would be located is one-half of the relation.

Finding the correct location for inserting and determining that it is a valid tuple to insert is the first step. Next, the tuple is inserted. However, inserting a new tuple in an ordered relation causes problems. First to maintain the ordering the tuple has a specific location where the tuple needs to be placed. If the block where the tuple should be placed contains room, the block only needs to be reorganized to move some of the tuples to make room at the proper location for the new tuple. However,

if the block does not have room for another tuple, either the tuples are shifted to make room in the block causing tuples to be shifted into another block forcing a reorganization of the remainder of the relation. If there is not room to insert the tuple and a reorganization is not desired, the tuple is written in an overflow area and a complete reorganization of the entire relation to include all tuple in the overflow area accomplished at some later time. If the overflow area is used, any retrieval algorithm using the ordering of the relation must also include some unordered portion to the processing to accommodate the overflow area.

Using the concept of maintaining the complete ordering of the relation, the performance model, using  $T_{sc}$  for the time necessary to shift a tuple within a block, is:

### Model U - 3

$$\begin{aligned}
 & T_c + T_d + T_{io} + \max \left| \begin{array}{c} (R/2) * T_{sc} \\ \text{or} \\ (((R/2)/b) * T_s) + ((R/2) * T_{io}) \end{array} \right| \\
 & + \left| \begin{array}{c} (R/2) * T_{sc} \\ \text{or} \\ (((R/2)/b) * T_s) + (2(R/2) * T_{io}) \end{array} \right|
 \end{aligned} \tag{170}$$

**6.1.1.4 Ordered - Indexed Insertion.** The ordered and indexed data structure requires two insertion operations. The first insertion adds the index value to the index. The second insertion adds the tuple in the correct location in the ordered relation. Therefore, this case uses the index to determine if the tuple to be inserted is a duplicate and the location of the insertion. Then, the insertion operation continues as in the previous case, unindexed-ordered relation.

The model combining the insertion in the index and an ordered relation is:

## Model U - 4

$$T_c + ((L_I + 1) * T_{ind}) + T_{io} + [(1 / (\lceil z/2 \rceil - 1)) * 2T_{ind}] + \begin{cases} (R/2) * T_{sc} \\ (((R/2)/b) * T_s) + [2(R/2) * T_{io}] \end{cases} \text{ or } (171)$$

### 6.1.2 Single Processor - Multiple Disk Insertion.

6.1.2.1 *Unordered - Unindexed Insertion.* The first step of the insertion is the select to insure that the tuple to be inserted is not a duplicate. Then, the new tuple can be inserted in any block that has space for the tuple. The only difference from the single processor - single disk unordered - unindexed insertion is that the disk seeks would be reduced during the duplicate check. Therefore, the performance model is:

## Model U - 5

$$T_c + T_d + T_{io} + \max \begin{cases} (R - 1) * T_{sc} \\ (((R/d) * T_{io}) - 1) \end{cases} \text{ or } + T_{sc} + T_{io} \quad (172)$$

6.1.2.2 *Unordered - Indexed Insertion.* The insertion uses the index to determine if the tuple to be inserted is a duplicate. The only difference using multiple disks versus the single disk - single processor case is that the index may be stored on more than one disk. However, retrieving each block of the index still requires a disk access and I/O because the next block to be retrieved can not be predetermined. Thus, the performance model equates to the single disk - single processor unordered - indexed insertion model.

6.1.2.3 *Ordered - Unindexed Insertion.* Assuming the relation is stored on more than one disk, the first step is to find the disk that has the proper insertion

location for the tuple to be inserted. Then the proper location within the disk is located. The expected value for locating the proper disk is .5 times the number of disks and .5 times the number of blocks contained on the disk. If the tuple is not a duplicate, then the blocks within the disk are reorganized to provide room for the insertion. This method may leave partial blocks on several disks. But, the insertion time is much less than if the entire relation following the insertion point must be reorganized. The performance model (using  $T_{sc}$  for the time to shift tuple within a block) is:

### Model U - 6

$$\begin{aligned}
 & T_c + T_d + T_{io} + .5d * T_{sc} + T_d + T_{io} + \max \left| \begin{array}{c} .5(R/d) * T_{sc} \\ \text{or} \\ (.5(R/d) * T_{io}) \\ +[(.5(R/d)/b) * T_s] \end{array} \right| \\
 & + \max \left| \begin{array}{c} +(.5(R/d) * T_{sc}) \\ \text{or} \\ (2(.5(R/d) - 1) * T_{io}) \end{array} \right| + T_{io} \quad (173)
 \end{aligned}$$

*6.1.2.4 Ordered - Indexed Insertion.* The ordered - indexed data structure requires two insertion operations. The first insertion adds the index value to the index. The second insertion adds the tuple in the correct location in the ordered relation. Therefore, this case first uses the index to determine if the tuple to be inserted is a duplicate and the location for the insertion of the tuple into the relation. Then the index is updated and the tuple inserted in the relation and the relation reorganized as necessary. Therefore, the performance model is a combination of the two previous cases. The performance model is:

## Model U - 7

$$T_c + ((L_I + 1) * T_{ind}) + T_{io} + [(1/(\lceil z/2 \rceil - 1)) * 2T_{ind}]$$

$$+ \max \left| \begin{array}{c} .5(R/d) * T_{sc} \\ \text{or} \\ ((.5(R/d)/b) * T_s) + [2(.5(R/d)) * T_{io}] \end{array} \right| \quad (174)$$

### 6.1.3 Multiple Processor - Single Disk Insertion.

6.1.3.1 *Unordered - Unindexed Insertion.* The unorganized data structure requires the entire relation to be scanned to determine if the tuple to be inserted is a duplicate. This is actually a selection operation with the writing of a block that contains the inserted tuple. Therefore, the performance model is a variation of the select model. The performance model for updating the unordered - unindexed relation in the multiple processor - single disk architecture is:

## Model U - 8

$$T_c + T_m + T_d + T_{io} + \max \left| \begin{array}{c} ((R/p) - 1) * T_{sc} \\ \text{or} \\ ([R/b] - 1) * T_s \\ + ((R - 1) * T_{io}) \end{array} \right| + T_{sc} + (p-1) * T_m + T_{bt} + T_d + T_{io} \quad (175)$$

6.1.3.2 *Unordered - Indexed Insertion.* This case does not present any variation from the single processor - single disk case. Because the multiple processors cannot anticipate which index block to retrieve next, only a single processor operates on the index. Therefore, refer to the single disk - single processor case for the performance model.

6.1.3.3 *Ordered - Unindexed Insertion.* This case uses the processors to scan the relation to find the proper insertion position. The only difference between

this case and the same insertion with the single disk - single processor case is that the multiple processors can be used to reduce the processing time of scanning the relation. Therefore, the performance model (using  $T_{sc}$  for the time necessary to shift tuples within a block) is:

### Model U - 9

$$\begin{array}{c}
 T_c + T_m + T_d + T_{io} + \max \left| \begin{array}{c} ((R/2)/p) * T_{sc} \\ or \\ (((R/2)/b) * T_s) + ((R/2) * T_{io}) \end{array} \right| \\
 + \max \left| \begin{array}{c} ((R/2)/p) * T_{sc} \\ or \\ (((R/2)/b) * T_s) + [2(R/2) * T_{io}] \end{array} \right|
 \end{array} \quad (176)$$

*6.1.3.4 Ordered - Indexed Insertion.* The ordered and indexed data structure requires two insertion operations. The first insertion adds the index value to the index. The second insertion adds the tuple in the correct location in the ordered relation. Therefore, this case first uses the index to determine if the tuple to be inserted is a duplicate. Then, the insertion operations can begin. The model combining the insertion in the index and an ordered relation is:

### Model U - 10

$$\begin{array}{c}
 T_{io} + [(1/(\lceil z/2 \rceil - 1)) * (2 * (T_{io} + T_d))] \\
 + ((.5(R)/b) * T_s) + [2(.5(R)) * T_{io}] \\
 + \max \left| \begin{array}{c} .5(R) * T_{sc} \\ or \\ +[(1/(\lceil z/2 \rceil - 1)) * T_{sc}] \end{array} \right|
 \end{array} \quad (177)$$

### 6.1.4 Multiple Processor - Multiple Disk Insertion.

**6.1.4.1 Unordered - Unindexed Insertion.** The main portion of the unordered - unindexed insertion is scanning the entire relation to determine if the tuple to be inserted is a duplicate. After the insertion is determined to be valid, a single block must be written with the new tuple. The performance model is:

### Model U - 11

$$T_c + T_m + T_d + T_{io} + \max \left| \begin{array}{l} ((R/p) - 1) * T_{sc} \\ \text{or} \\ [((R/d)b) - 1] * T_s \\ + (((R/d) - 1) * T_{io}) \end{array} \right| + T_{sc} + (p-1)*T_m + T_{bt} + T_d + T_{io} \quad (178)$$

**6.1.4.2 Unordered - Indexed Insertion.** The index controls the processing since it is used to determine if the tuple is a duplicate. Therefore, only the updating of the index and reading and insertion of the tuple in a block can be overlapped. Thus, the performance model portrays one processor checking the index for duplicates and then a processor inserting the value in the index and one processor inserting the tuple in the appropriate block. The performance model is:

### Model U - 12

$$T_c + ((L_I + 1) * T_{ind}) + \max \left| \begin{array}{l} T_{io} + [(1/(\lceil z/2 \rceil - 1)) * 2T_{ind}] \\ \text{or} \\ T_d + T_{io} + T_{sc} + T_{io} \end{array} \right| \quad (179)$$

**6.1.4.3 Ordered - Unindexed Insertion.** The ordered insertion requires the correct location for the insertion to be found. This also determines if the tuple to be inserted is a duplicate. Then the blocks on the disks where the insertion location is located are reorganized to make room for the insertion. The multiple processors concurrently scan for the insertion location. The reorganization involves only one disk and one processor. Therefore, the performance model (using  $T_{shift}$  = the time necessary to shift tuple within a block) is:

### Model U - 13

$$\begin{aligned}
 T_c + T_m + T_d + T_{io} + (.5d/p * T_{sc}) + \max & \left| \begin{array}{l} (.5(R/d)/p) * T_{sc} \\ \text{or} \\ ((.5(R/d)/b) * T_s) + (.5(R/d) * T_{io}) \end{array} \right| \\
 + \max & \left| \begin{array}{l} (.5(R/d)/p) * T_{sc} \\ \text{or} \\ ((.5(R/d)/b) * T_s) + (.5(R/d) * T_{io}) \end{array} \right| + T_{io} \quad (180)
 \end{aligned}$$

**6.1.4.4 Ordered - Indexed Insertion.** This case combines the previous two cases of inserting using an index and inserting the tuple into an ordered relation. The index is used to determine the insertion location and if the tuple is a duplicate. The relation is then reorganized as necessary to make room for the insertion of the tuple. The model combining the two previous cases is:

### Model U - 14

$$\begin{aligned}
 T_c + ((L_I + 1) * T_{ind}) + \max & \left| \begin{array}{l} T_{io} + [(1/(\lceil z/2 \rceil - 1)) * 2T_{ind}] \\ \text{or} \\ .5(R/d) * T_{sc} \\ \text{or} \\ ((.5(R/d)/b) * T_s) + [2(.5(R/d)) * T_{io}] \end{array} \right| \quad (181)
 \end{aligned}$$

## 6.2 Deleting a tuple.

The deletion of a tuple from a relation requires the appropriate tuple(s) to be found and removed. The deletion criteria could be a range for an attribute or a single key value. For this discussion, it is assumed that only a single tuple will be deleted. The deletion of additional tuples does not alter the processing except that it may require additional block writes depending upon the data structure. Therefore, the models developed here assume only deleting a single tuple per deletion operation. The deletion is basically a select operation. This is similar to a insert; except, the

delete never has to worry about overflowing a block causing a reorganization of the relation. Therefore, all of the following performance models closely resemble the corresponding select performance models and also are similar to the insert operations modeled above.

### 6.2.1 Single Processor - Single Disk Deletion.

6.2.1.1 *Unordered - Unindexed Deletion.* The deletion of a tuple from a unordered - unindexed relation first requires the proper tuple to be found (a select operation). After a tuple is found, the tuple is removed from the block where it was found and the block written back on the disk. However, the remaining portion of the relation must be scanned because there is no guarantee this was the only tuple to be deleted.

The performance models for deleting a tuple from the relation R is:

#### Model U - 15

$$T_c + T_d + T_{io} + \max \left| \begin{array}{c} (R - 1) * T_{sc} \\ \text{or} \\ (((R/b) - 1) * T_s) + ((R - 1) * T_{io}) \end{array} \right| + T_{sc} + T_{io} \quad (182)$$

6.2.1.2 *Unordered - Indexed Deletion.* The indexed - unordered deletion is basically a select with a rewrite of the block where the tuple was deleted and an update of the index. Therefore, the performance model is:

#### Model U - 16

$$T_c + ((L_I + 1) * T_{ind}) + T_{io} + T_d + T_{io} + T_{sc} + T_{io} \quad (183)$$

6.2.1.3 *Ordered - Unindexed Deletion.* The deletion of a tuple from an ordered relation is controlled by where the tuple is located in the sequential order. Since the tuple location is not known, the expected value of finding a single tuple is

used (.5R). After the block with the proper tuple is found, the tuple is deleted and the block rewritten. This process leaves some extra space in the block where the tuple was removed. However, it does not alter the sequential order of the relation. The following model reflect the time necessary to execute a deletion of a tuple from a ordered relation.

### Model U - 17

$$T_c + T_d + T_{io} + \max \left| \begin{array}{c} (R/2) * T_{sc} \\ \text{or} \\ (((R/2)/b) * T_s) + ((R/2) * T_{io}) \end{array} \right| + T_{io} \quad (184)$$

**6.2.1.4 Ordered - Indexed Deletion.** The deletion of a tuple from an ordered - indexed relation requires two separate deletions—deleting the tuple and deleting the index entry for the tuple. First, the index is used to retrieve a TID. This TID is then used to retrieve the block that contains the tuple and the tuple is deleted. Also, the TID must be deleted from the index. Since the block where the tuple is deleted is not reorganized, this is equal to the performance model for deleting a tuple from an indexed - unordered relation. Therefore, refer to the previous section for the indexed deletion for the performance model.

### 6.2.2 Single Processor - Multiple Disk Deletion.

**6.2.2.1 Unordered - Unindexed Deletion.** The deletion of a tuple first finds the appropriate tuple(s). Then the tuple is removed and the block is written back to the disk. Therefore, the performance model is:

### Model U - 18

$$T_c + T_d + T_{io} + \max \left| \begin{array}{c} (R - 1) * T_{sc} \\ \text{or} \\ (((R/d) * T_{io}) - 1) \end{array} \right| + T_{sc} + T_{io} \quad (185)$$

**6.2.2.2 Unordered - Indexed Deletion.** This case may utilize more than one disk to store the index. However, each block of the index still requires a disk access and I/O because the next block to be processed is not predetermined. Thus, this deletion still requires the same time to execute as the single processor - single disk case.

**6.2.2.3 Ordered - Unindexed Deletion.** The deletion is a select followed by processing the disk where the tuple to be deleted is located. After the block with the proper tuple is found, the tuple is deleted and the block rewritten. This process leaves some extra space in the block where the tuple was removed. However, it does not alter the sequential order of the relation. The performance model is:

#### Model U - 19

$$T_c + T_d + T_{io} + .5d * T_{sc} + T_d + T_{io} + \max \left( \begin{array}{l} .5(R/d) * T_{sc} \\ \text{or} \\ (.5(R/d) * T_{io}) \\ + [(.5(R/d)/b) * T_s] \end{array} \right) + T_{io} \quad (186)$$

**6.2.2.4 Ordered - Indexed Deletion.** The deletion of a tuple from an ordered - indexed relation requires two separate deletions. First, the index is used to retrieve the TID. This TID is then used to retrieve the block that contains the tuple and the tuple is deleted. Also, the TID must be deleted from the index. Since the block where the tuple is deleted is not reorganized, this is the same performance model as the performance model for deleting a tuple from an indexed - unordered relation. Therefore, refer to the previous case for the indexed deletion for the performance model.

#### 6.2.3 Multiple Processor - Single Disk Deletion.

**6.2.3.1 Unordered - Unindexed Deletion.** The deletion of a tuple from a unordered - unindexed relation with multiple processors and a single disk is es-

sentially the same as the single processor - single disk processing for the same case except the processing may be done concurrently by the processors. Therefore, the performance model is:

### Model U - 20

$$T_c + T_m + T_d + T_{io} + \max \left| \begin{array}{l} ((R/p) - 1) * T_{sc} \\ \text{or} \\ (((R/b) - 1) * T_s) \\ + ((R - 1) * T_{io}) \end{array} \right| + T_{sc} + (p - 1) * T_m + T_{bt} + T_d + T_{io} \quad (187)$$

*6.2.3.2 Unordered - Indexed Deletion.* This case does not present any variation from the single processor - single disk case because the multiple processors cannot anticipate which index block to retrieve next. This means that only a single processor can operate on the index. Therefore, refer to the single disk - single processor case for the performance model.

*6.2.3.3 Ordered - Unindexed Deletion.* The deletion of a tuple from an ordered relation is controlled by where the tuple is located in the sequential order. Since the tuple location is not known, the expected value of finding a single tuple is used (.5R). After the block with the proper tuple is found, the tuple is deleted and the block rewritten. The performance model using multiple processors is:

### Model U - 21

$$T_c + T_m + T_d + T_{io} + \max \left| \begin{array}{l} ((R/2)/p) * T_{sc} \\ \text{or} \\ (((R/2)/b) * T_s) + ((R/2) * T_{io}) \end{array} \right| + T_{io} \quad (188)$$

*6.2.3.4 Ordered - Indexed Deletion.* This deletion requires two deletion - deleting the value from the index and deleting the tuple. However, the deletion of the tuple from the ordered relation does not alter the ordering of the relation.

Therefore, the deletion of the tuple is no different than deleting the tuple from the unordered relation. Therefore, the performance model equates to the performance model for the unordered - indexed deletion case.

#### *6.2.4 Multiple Processor - Multiple Disk Deletion.*

*6.2.4.1 Unordered - Unindexed Deletion.* The deletion in the multiple processor - multiple disk environment allows the processors to concurrently scan for the tuple to be deleted. When the tuple to be deleted is found the tuple is removed and the block written back on disk. However, the remaining portion of the relation must be scanned because there is no guarantee this was the only tuple that satisfied the selection condition of the deletion. Therefore, the performance model is a selection plus a block write.

#### **Model U - 22**

$$T_c + T_m + T_d + T_{io} + \max \left( \begin{array}{l} ((R/p) - 1) * T_{sc} \\ \text{or} \\ [((R/d)b) - 1] * T_s \\ + ((R/d) - 1) * T_{io} \end{array} \right) + T_{sc} + (p-1)*T_m + T_{bt} + T_d + T_{io} \quad (189)$$

*6.2.4.2 Unordered - Indexed Deletion.* Since the index controls the processing, only the updating of the index and reading and insertion of the tuple in a block can be overlapped. Thus, the performance model portrays one processor processing the index. Then one processor can retrieve and delete the tuple from the block where it is located and the processor processing the index can update the index.

#### **Model U - 23**

$$T_c + ((L_I + 1) * T_{ind}) + \max \left( \begin{array}{l} T_{sc} + T_{io} \\ \text{or} \\ T_d + T_{io} + T_{sc} + T_{io} \end{array} \right) \quad (190)$$

This model shows the overlapping updates. The deletion of the tuple requires more time than the deletion of the index entry because the block with the tuple has to be retrieved and then written. Whereas, the index block has already been read for processing, so it only requires the update to be made and the index block written. Therefore, the model actually is:

### **Model U - 24**

$$T_c + ((L_I + 1) * T_{ind}) + T_d + T_{io} + T_{sc} + T_{io} \quad (191)$$

*6.2.4.3 Ordered - Unindexed Deletion.* This deletion uses the concept of scanning the first block of each disk to more quickly find the tuple to be deleted. The deletion then continues in the same manner as the unordered case. Therefore, the performance model is:

### **Model U - 25**

$$T_c + T_m + T_d + T_{io} + (.5d/p * T_{sc}) + \max \left| \begin{array}{c} (.5(R/d)/p) * T_{sc} \\ \text{or} \\ ((.5(R/d)/b) * T_s) + (.5(R/d) * T_{io}) \end{array} \right| + T_{io} \quad (192)$$

*6.2.4.4 Ordered - Indexed Deletion.* The deletion of a tuple from an ordered - indexed relation requires two separate deletions. First, the index is used to retrieve a TID. This is used to retrieve the block that contains the tuple and the tuple is deleted. Also, the TID must be deleted from the index. Since the block where the tuple is deleted is not reorganized, this is the same performance model as the performance model for deleting a tuple from an indexed - unordered relation. Therefore, refer to the previous case for the indexed deletion for the performance model.

### **6.3 Modifying a tuple**

The modification of a tuple can be thought of as a deletion of an old tuple and an insertion of a new tuple. However, this is not totally true. For the case where the deletion changes the key attribute(s) of the tuple, this is true but if the key attribute(s) are unchanged in the modification, not all the integrity checking of the insertion is necessary.

For the case of the key attribute not being changed, the performance is closely modeled by the deletion models of the previous section. Therefore, the models presented in this section deal with the more general case of a modification that might include the key attribute(s). Since the key attribute may be changed, integrity checking will be necessary. Therefore, there will three operations occurring: the integrity check for introduction of a duplicate tuple, the insertion of the new tuple, and deletion of the old tuple.

#### **6.3.1 Single Processor - Multiple Disk Modification.**

**6.3.1.1 Unordered - Unindexed Modification.** The modification of a tuple in the unordered - unindexed data structure may require scanning the relation twice. The unordered - unindexed data structure provides no means of integrity checking other than scanning the entire relation. The integrity check must be complete before inserting or deleting the tuple can safely be accomplished. Therefore, the entire relation must first be scanned, then the deletion and insertion can be executed. However, this process is improved by identifying the block where the tuple to be deleted is located. Then only this block must be read to accomplish the deletion. And the tuple to be inserted can be placed in the place where the tuple was deleted. Thus, by scanning the entire unordered - unindexed relation once the location of the deletion and the integrity checking is accomplished. The performance model is:

## Model U - 26

$$T_c + T_d + T_{io} + \max \left| \begin{array}{l} (R - 1) * T_{sc} \\ \text{or} \\ (((R/b) - 1) * T_s) \\ + ((R - 1) * T_{io}) \end{array} \right| + T_{sc} + T_{io} + T_d + T_{io} + T_{sc} + T_{io} \quad (193)$$

*6.3.1.2 Unordered - Indexed Modification.* The indexed data structure allows the index to be searched to determine if the tuple to be inserted is a duplicate. This assumes that the index was constructed for the attribute(s) that form the key for the relation. If the index is constructed with an attribute(s) that form only a portion of the key, then the index would identify the tuple(s) to be retrieved to be examined to determine if they are duplicates of the tuple being inserted. Then the tuples are retrieved and compared to the tuple to be inserted to insure they are not duplicates.

The next step after the new tuple is determined not to be a duplicate is delete the old tuple and insert the new tuple. Since the relation is not ordered, the insertion can occur in the same location as the deletion. The index then must be updated. Updating the index includes adding the new value and deleting the old value. This requires the index to be processed twice and may include a split when adding the new value. The performance model is:

## Model U - 27

$$T_c + ((L_i + 1) * T_{ind}) + T_{io} + T_d + T_{io} + T_{sc} + T_{io} + ((L_i + 1) * T_{ind} + T_{io} + [(1 / (\lceil z / 2 \rceil - 1)) * 2T_{ind}]) \quad (194)$$

*6.3.1.3 Ordered - Unindexed Modification.* The modification of the ordered relation requires two phases: the integrity check phase and the modification phase. The first phase must determine if the new tuple would duplicate an existing tuple. The relation is scanned until the appropriate location in the relation for the

insertion is located. If no duplicate exists the insertion can be accomplished. However, to make room for the insertion all the following data must be shifted until a block contains extra space.

The modification also requires a deletion. The deletion cannot occur until it is confirmed that the insertion will not violate the integrity of the database. Therefore, two situations could occur involving the location of the deletion and the insertion. The first would be that the location of the deletion was before the location of the insertion. This case would require the location of deletion to be noted during the scan insuring the data integrity. The scan of the relation then continue to find the insertion location for the new tuple. When the scan finds the insertion location and confirms that the new tuple is not a duplicate, the deletion can be completed and tuples shifted up to the insertion point to make room for the insertion.

The other case is where the insertion location occurs before the deletion location. This would require the insertion to be accomplished and the data shifted to make room for the insertion until the location of the deletion is found. At this point, the deletion would create the space necessary for the shifted data, stopping the shifting of data to maintain the proper ordering. This obviously assumes that the blocks are all full and an overflow area is not used. If an overflow area was utilized, the shifting would not occur but both the blocks that contained the insertion and deletion locations would be modified to reflect the modifications.

The performance models for modifying an ordered relation require several assumptions. The first is that relation will be reorganized at the time of the action. The next assumption states that the expected value of finding the insertion location for integrity checking requires scanning one half the relation. It is also assumed that deletion point is .5 of one of the halves of the relation away from the insertion point. This assumption allows the approximation of the number of blocks necessary to be reorganized during the modification. Using these assumptions, the performance models are (using  $T_{sc}$  for the time necessary to shift tuples within a block):

## Model U - 28

$$T_c + T_d + T_{io} + \max \left| \begin{array}{c} (R/2) * T_{sc} \\ \text{or} \\ (((R/2)/b) * T_s) + ((R/2) * T_{io}) \end{array} \right| \quad (195)$$

$$+ \max \left| \begin{array}{c} (R/4) * T_{sc} \\ \text{or} \\ (((R/4)/b) * T_s) + [2(R/4) * T_{io}] \end{array} \right|$$

*6.3.1.4 Ordered - Indexed Modification.* The ordered and indexed data structure requires two modification operations. The first modification changes the index. The second modification alters the tuple in the ordered relation. Therefore, this case first uses the index to determine if the new tuple is a duplicate. Then the modification operations can begin. The model combining the modification of the index and an ordered relation is:

## Model U - 29

$$T_c + ((L_I + 1) * T_{ind}) + T_{io} + [(1/(\lceil z/2 \rceil - 1)) * 2T_{ind}]$$

$$+ ((L_I + 1) * T_{ind}) + T_{io} + \max \left| \begin{array}{c} (R/4) * T_{sc} \\ \text{or} \\ (((R/4)/b) * T_s) \\ + [2(R/4) * T_{io}] \end{array} \right| \quad (196)$$

*6.3.2 Single Processor - Multiple Disk Modification.*

*6.3.2.1 Unordered - Unindexed Modification.* The multiple disk environment takes the performance model for the single processor - single disk environment and reduces the performance by eliminating the disk accesses time. Therefore, the performance model is:

### Model U - 30

$$T_c + T_d + T_{io} + \max \left| \begin{array}{c} (R - 1) * T_{sc} \\ \text{or} \\ (((R/d) * T_{io}) - 1) \end{array} \right| + T_{sc} + T_{io} + T_d + T_{io} + T_{sc} + T_{io} \quad (197)$$

**6.3.2.2 Unordered - Indexed Modification.** This case may utilize more than one disk to store the index. However, the disk can not be positioned in advance to read the next block because the next block is determined by processing the current block. Therefore, the advantage of using multiple disks to reduce the number of disk accesses is eliminated. Thus, this equates to the model for the single processor - single disk indexed - unordered modification case.

**6.3.2.3 Ordered - Unindexed Modification.** This case requires the correct disk to be located and then the conditions processing described previously applies. The one exception to the reorganization stated above, for the ordered - indexed modification with a single processor - single disk case, is that only the blocks contained within the disk would have to be reorganized. For the modification, it is assumed that the insertion and deletion are different disks requiring the reorganization of the data on two disks. Also, it is assumed that half of the remaining disks must be searched to find the location of either the insertion or deletion. Using this criteria the performance model is:

### Model U - 31

$$T_c + T_d + T_{io} + .5d * T_{sc} + T_d + T_{io} + \max \left| \begin{array}{c} .5(R/d) * T_{sc} \\ \text{or} \\ (.5(R/d) * T_{io}) \\ + [(.5(R/d)/b) * T_s] \end{array} \right|$$

$$\begin{aligned}
& + \max \left| \begin{array}{c} +(.5(R/d) * T_{sc}) \\ \text{or} \\ (2(.5(R/d) - 1) * T_{io}) \end{array} \right| + T_{io} + \max \left| \begin{array}{c} .25(R/d) * T_{sc} \\ \text{or} \\ (.25(R/d) * T_{io}) \\ +[(.25(R/d)/b) * T_s] \end{array} \right| \\
& + \max \left| \begin{array}{c} +(.25(R/d) * T_{sc}) \\ \text{or} \\ (2(.25(R/d) - 1) * T_{io}) \end{array} \right| + T_{io} \quad (198)
\end{aligned}$$

**6.3.2.4 Ordered - Indexed Modification.** The ordered and indexed data structure requires two modification operations. The first modification changes the index. The second modification alters the tuple in the ordered relation. Therefore, this case first uses the index to determine if the new tuple is a duplicate. Then, the modification operations can begin. The model combining the modification of the index and an ordered relation is:

### Model U - 32

$$T_c + ((L_I + 1) * T_{ind}) + T_{io} + [(1/([z/2] - 1)) * 2T_{ind}]$$

$$\begin{aligned}
& +((L_I + 1) * T_{ind}) + T_{io} + \max \left| \begin{array}{c} .5(R/d) * T_{sc} \\ \text{or} \\ ((.5(R/d)/b) * T_s) \\ +[2(.5(R/d)) * T_{io}] \end{array} \right| \\
& + \max \left| \begin{array}{c} .5(R/d) * T_{sc} \\ \text{or} \\ ((.5(R/d)/b) * T_s) + [2(.5(R/d)) * T_{io}] \end{array} \right| \quad (199)
\end{aligned}$$

### 6.3.3 Multiple Processor - Single Disk Modification.

*6.3.3.1 Unordered - Unindexed Modification.* The multiple processor environment allows concurrent scanning of the relation. However, the single disk may constrain the processing. Therefore, the following performance model modifies the single processor modification by including the multiple processor capability. With this the performance model is:

### Model U - 33

$$T_c + T_m + T_d + T_{io} + \max \left| \begin{array}{c} ((R/p) - 1) * T_{sc} \\ \text{or} \\ ([R/b] - 1) * T_s + ((R - 1) * T_{io}) \end{array} \right| + T_{sc} + (p - 1) * T_m + T_{bt} + T_d + T_{io} + T_{sc} + T_{io} \quad (200)$$

*6.3.3.2 Unordered - Indexed Modification.* This case does not present any variation from the single processor - single disk indexed - unordered case. Because the multiple processors cannot anticipate which index block to retrieve next, only a single processor operates on the index. Therefore, refer to the single disk - single unordered - indexed processor case for the performance model.

*6.3.3.3 Ordered - Unindexed Modification.* This case is the same as the previous unordered - unindexed case except that now the expected search space will be reduced by the ordering of the relation. But, this also causes the insertion place of a new tuple to be reorganized to make room for the tuple (assuming the modification changes the key of the relation). This performance model is also very similar to the single processor - single disk case except the processing capacity is increased with the multiple processors. Using this the performance model is:

### Model U - 34

$$\begin{array}{c|c}
 T_c + T_m + T_d + T_{io} + \max & \left. \begin{array}{c} ((R/2)/p) * T_{sc} \\ \text{or} \\ (((R/2)/b) * T_s) + ((R/2) * T_{io}) \end{array} \right| \\
 \hline
 & (201)
 \end{array}$$

$$\begin{array}{c|c}
 + \max & \left. \begin{array}{c} ((R/4)/p) * T_{sc} \\ \text{or} \\ (((R/4)/b) * T_s) + [2(R/4) * T_{io}] \end{array} \right| \\
 \hline
 &
 \end{array}$$

**6.3.3.4 Ordered - Indexed Modification.** This model combines the use of the index to find the proper location of the previous indexed case and the reorganization of the relation in inserting the tuple. Therefore, the combination of these two models that modify the ordered - indexed relation is:

### Model U - 35

$$\begin{array}{c|c}
 T_{io} + [(1/(\lceil z/2 \rceil - 1)) * (2 * (T_{io} + T_d))] \\
 + ((L_I + 1) * (T_d + T_{io})) + T_{io} \\
 + (.25(R/b) * T_s) + [2(.25(R)) * T_{io}] \\
 \hline
 T_c + ((L_I + 1) * T_{ind}) + \max & \left. \begin{array}{c} \text{or} \\ .25(R) * T_{sc} \end{array} \right| \\
 \hline
 + \max & \left. \begin{array}{c} \text{or} \\ +[(1/(\lceil z/2 \rceil - 1)) * T_{sc}] + ((L_I + 1) * T_{sc}) \end{array} \right| \\
 \hline
 & (202)
 \end{array}$$

### 6.3.4 Multiple Processor - Multiple Disk Modification.

**6.3.4.1 Unordered - Unindexed Modification.** The multiple processor - multiple disk environment provides the capability to handle the modification (assuming a deletion and an insertion) two ways. The first method would devote a portion of the processors to each action. However, this could be a problem if the

insertion was a duplicate, invalidating the insertion then the deletion would have to be unaccomplished if it had been completed.

The second method and the method modeled here uses all of the processors for the operation being performed at that time. This also reduces the contention for retrieving from a specific disk. Therefore, the performance model reflects the combination of inserting and deleting a tuple. In this case, that means scanning the entire relation to find the tuple to be deleted and to insure the insertion is not a duplicate. Using this, the performance model is:

### Model U - 36

$$T_c + T_m + T_d + T_{io} + \max \left| \begin{array}{c} ((R/p) - 1) * T_{sc} \\ \text{or} \\ (((R/d)b) - 1) * T_s + (((R/d) - 1) * T_{io}) \end{array} \right| + T_{sc} + (p - 1) * T_m + T_{bt} + T_d + T_{io} + T_{sc} + T_{io} \quad (203)$$

*6.3.4.2 Unordered - Indexed Modification.* Since the index controls the processing, only the updating of the index and reading and insertion of the tuple in a block can be overlapped. Thus, the performance model portrays one processor checking the index for duplicates and then a processor inserting the value in the index and one processor inserting the tuple in the appropriate block.

### Model U - 37

$$T_c + ((L_I + 1) * T_{ind}) + \max \left| \begin{array}{c} T_{io} + [(1/(\lceil z/2 \rceil - 1)) * 2T_{ind}] + ((L_I + 1) * T_{ind}) + T_{io} \\ \text{or} \\ T_d + T_{io} + T_{sc} + T_{io} \end{array} \right| \quad (204)$$

*6.3.4.3 Ordered - Unindexed Modification.* The ordered relation allows less of the relation to be searched to find the insertion and deletion location. The multiple processors are used to scan the initial block of each disk to determine the

disks that contain the action location. Then the insertion or deletion can continue. Using the assumptions about the ordered relation structure contained in the single processor - single disk ordered - unindexed case, the performance model is:

### Model U - 38

$$\begin{aligned}
 & T_c + T_m + T_d + T_{io} + (.5d/p * T_{sc}) + \max \left| \begin{array}{l} (.5(R/d)/p) * T_{sc} \\ \text{or} \\ ((.5(R/d)/b) * T_s) \\ + (.5(R/d) * T_{io}) \end{array} \right| \\
 & + \max \left| \begin{array}{l} (.5(R/d)/p) * T_{sc} \\ \text{or} \\ ((.5(R/d)/b) * T_s) \\ + (.5(R/d) * T_{io}) \end{array} \right| + T_{io} \max \left| \begin{array}{l} (.25(R/d)/p) * T_{sc} \\ \text{or} \\ ((.25(R/d)/b) * T_s) \\ + (.25(R/d) * T_{io}) \end{array} \right| \\
 & + \max \left| \begin{array}{l} (.25(R/d)/p) * T_{sc} \\ \text{or} \\ ((.25(R/d)/b) * T_s) \\ + (.25(R/d) * T_{io}) \end{array} \right| + T_{io} \quad (205)
 \end{aligned}$$

*6.3.4.4 Ordered - Indexed Modification.* This case uses the index for identifying the deletion and insertion location. The index processing is restrictive. Therefore, only the index and relation updating is overlapped. Thus, the performance model is:

### Model U - 39

$$\begin{aligned}
 & T_{io} + [(1/(\lceil z/2 \rceil - 1)) * 2T_{ind}] + +((L_I + 1) * T_{ind}) + T_{io} \\
 & + \max \left| \begin{array}{l} .5(R/d) * T_{sc} \\ \text{or} \\ ((.5(R/d)/b) * T_s) + [2(.5(R/d)) * T_{io}] \end{array} \right| \quad (206)
 \end{aligned}$$

#### *6.4 Summary*

All of the models presented for the update actions closely resemble the select performance models due to the demand of not introducing duplicate tuples and/or finding the proper location or tuple for processing. However, the more complex or optimized data structures now require maintenance to maintain the proper structure. For the case of the indexed structures, this may include maintaining additional indexes on non-key attributes. The models then must be extended to add the additional index updates. In each model there is a segment that provides the index update, this would just be repeated for the number of indexes maintained for the given relation. For the ordered data structures, the structure maintenance required physically moving tuples to maintain the ordering. Therefore, the performance models for updates presented two aspects – the update action of relation and maintaining the data structure as necessary.

## *VII. Single Query Step Model Results*

The analytical models provided in the previous chapters do not prove that one algorithm or data structure is the "best". However, the models provide the opportunity to predict the results of various workload parameters. The first step in using the models for performance evaluation is determining the validity of the models. To prove the validity for each of the approximately 200 models would require each model to be actually implemented and the results compared. This is infeasible because of the volume of models and the lack of appropriate hardware. Therefore, the remaining method for determining the validity of the models is to compare the results of the models with accepted results. This method was used to validate the concept of the models by comparing the results with the models presented by Hawthorn and DeWitt [33] (where the models portrayed the same operation and same method of completing the operation). The results presented by Hawthorn and DeWitt have been validated by experimental benchmarking [9]. The results of Hawthorn and DeWitt have also become a "standard" referenced by many [5,7,36,64,67,84]. Therefore, the models have been compared to an accepted reference as appropriate to provide validation.

The analytical models presented are tools to assist in the design of a query processor. Since, the models contain so many variables, it is very difficult to analytically compare all models to determine the best model for a given application. And in designing a database machine, some of the parameters will vary for different applications of the same type processing causing the outcome to be different. Therefore, the results presented in this chapter are to be used as a guideline for developing the similar relationships for their performance parameters. This assumes that some of the performance parameters, such as, time to read or write a block from disk, the time seek a track, time to scan a block by the processor, etc., would be fixed and the

$p_b$	- blocks of memory per processor (50)
$b$	- blocks per track on the disk (20)
$T_c$	- time to compile query (60ms)
$T_m$	- time to send a message to/from back-end (2ms)
$T_d$	- average disk access time (39ms)
$T_s$	- seek time of one track on the disk (10ms)
$T_{io}$	- block read/write time (17ms)
$T_{sc}$	- time to scan block (10ms)
$T_{bt}$	- time to send block to back-end (7ms)
$T_b$	- time to process block with complex operation, like join (95ms)
$T_{ind}$	- time to fetch and examine an index page (66ms)
$R$	- number of blocks in relation R
$S$	- number of blocks in relation S
$B$	- number of bytes per block (13030)
$v$	- attribute size (10)
$r$	- tuple size (100 bytes)
$s$	- tuple size (100 bytes)
$in$	- index size in bytes (6 bytes)

Table 3. Performance Model Workload Parameters

relationship to be studied in the number of resources could best be used to solve a range of problems.

The following sections describe the performance patterns of the various data structures and algorithms for various workload environments. The presentation illustrates the various conditions graphically. The results presented were computed by programming each model as a separate function. This allows an individual or group of models to be compared for a given workload condition. Table 3 provides the values used for evaluating the performance time of the operators. The time parameters (fixed performance parameters described above) are based upon the parameters presented by Hawthorn and DeWitt [33].

### 7.1 Select

The select models were broken into categories, FT and MT. The FT cases are special cases of the MT cases. They present cases that are very difficult to

portray in the course environment of the MT cases. The queries that are looking for a single tuple as a response are the main concern of many databases. A bank database containing account balances or retrieving a person's name from a personal database given the person's ID number are examples of this type of query requirement. Therefore, the FT case presents a special situation that is normally overlooked in the interest of simplicity, resulting in a lack of completeness in evaluating all cases.

The focus of this investigation is to use multiple processors and parallel processing to improve the performance of database retrieval operations. The select operator has been shown to adapt easily to parallel processing (see Chapter II). The use of parallel processing does provide performance improvement to perform the select but the performance of the select may be hampered by the speed that data is provided to the processors for processing. Therefore, the secondary storage retrieval of the relations becomes the controlling factor. Several techniques have been suggested for improving the select operator [2,3,4,7,8,15,21,23,25,30,32,33,34,40,43,50,56,60,62] and no significant improvement seems feasible over the many specialized techniques and hardware presented. However, the effects of the data structure on the performance of the single item retrievals of the FT type transaction have been ignored. Therefore, Figure 22 presents the results of various select models for the multiple processor-multiple disk environment. The number of processors and disk selected for presentation in Figure 22 was arbitrary because the purpose of this evaluation and presentation was to illustrate the trend of the performance time as the input relation size increases but the expected output is limited to a few tuples.

Figure 22 shows the effect of increasing the input relation size on the completion of the selection operation. Figure 23 illustrates the same situation for the MT case, using a selectivity factor to compute the size of the results. This graph portrays that under different conditions, different data structures provide the best performance in performing the selection operation. Thus, it can be seen that varying the workload of the system may effect the performance of the system. This places a demand on

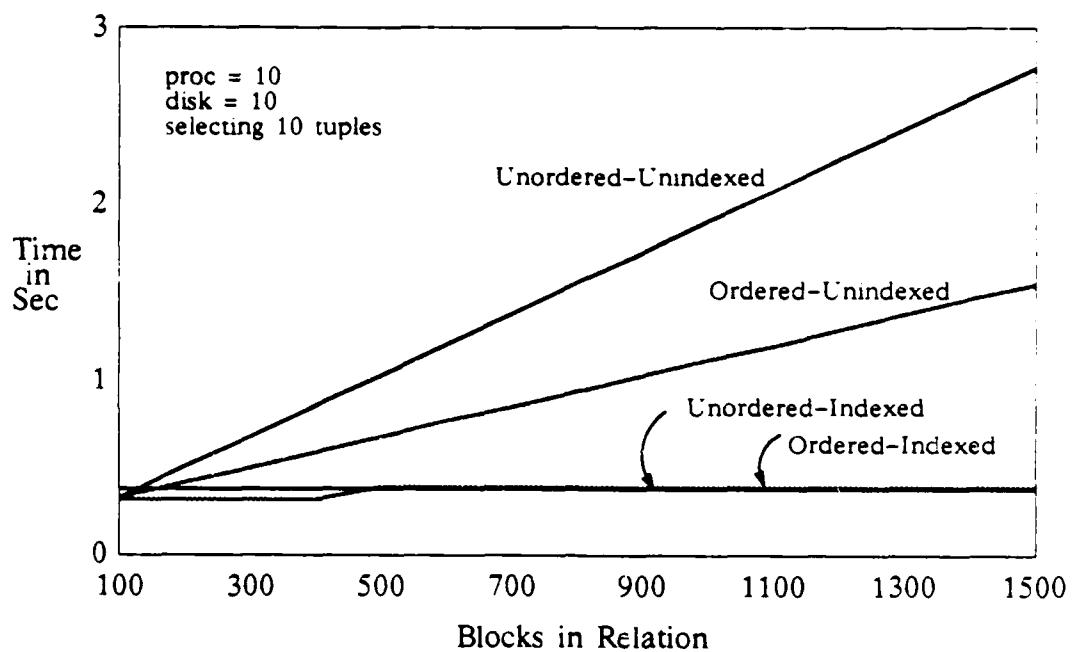


Figure 22. Multiple Processor-Disk Performance for the FT Select Operation

the database designer to define the requirements of the system. If the majority of the retrievals required can be predefined, the best data structure for that type of retrieval may be used in building the system. However, further investigations involving the performance of other operations are necessary to determine the impact of using the data structure most favorable for the selection operation.

## 7.2 Project

The projection operation models presented in Chapter IV reflect that data structure does not have an impact on the projection operation. Therefore, the performance results shown in Figure 24 for the multiple processor-multiple disk case show only two cases: projection-no duplicate removal and projection-with duplicate removal. It is obvious that if no duplicates are introduced or no concern about duplicates is given, the performance of the project is much better. Extending this, Figure 25 shows the effects of further distributing the storage and processing of the relation to complete the projection operation. Note that the disk and processors are

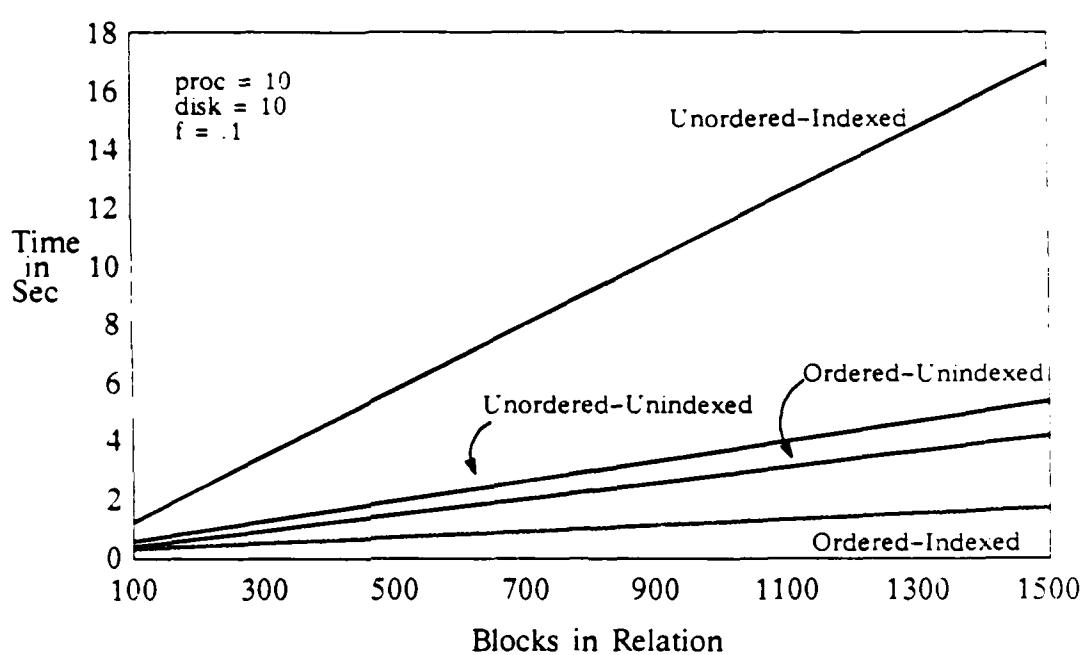


Figure 23. Multiple Processor-Multiple Disk Performance for the MT Select Operation

increased together because having a much greater number of one or the other significantly increases the performance capability of the operation. Therefore, the best storage-processor ratio is to provide at a minimum one disk for each processor. Since any less storage capability includes contention time with the already slow I/O time, a maximum number of disk units that can be efficiently used is difficult to determine without comparing the time requirement for the processor operation and the disk parameters. By using two disks per processor there is no contention for the disk to be writing at the same time it is reading data. Using any more disks per processor increases the data bandwidth, but may not produce significant performance time increases.

The project operation performance is data structure independent in a horizontally partitioned database. Therefore, the projection operation has no impact in determining which data structure is "best". In fact, the data structure independence of the projection operation implies that the projection and selection operations could

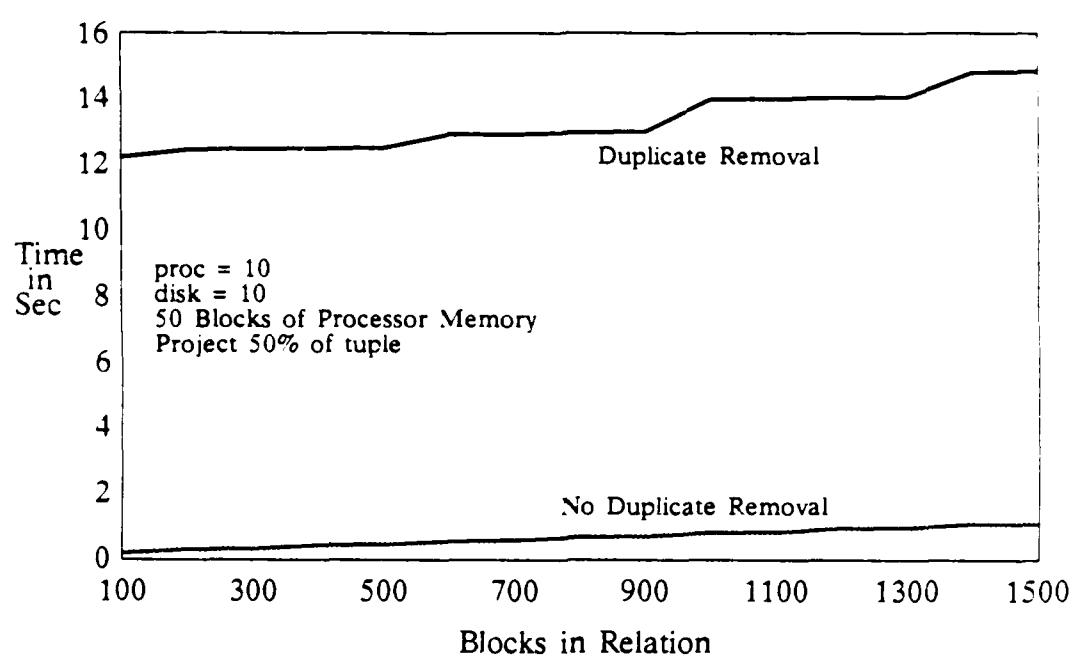


Figure 24. Multiple Processor-Multiple Disk Performance for the Project Operation

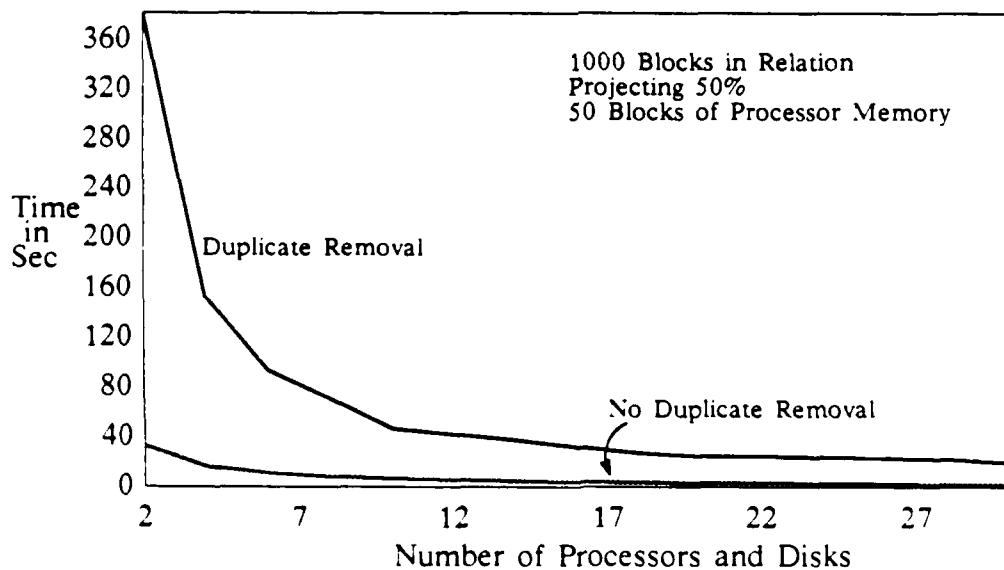


Figure 25. The Performance Effect of Increasing Resources on the Project Operation

be combined into a single operation for applications that require both operations in performing the user query.

### 7.3 Join

The join (equi-join) performance models, like the selection operation, are sensitive to the data structure of the input relation. The obvious example of the data structure effecting the performance of the join is when both relations are already ordered. In this case, the two relations are joined in a merge-type join that requires each relation to only be read once. However, if the relations are not sorted, the relations must first be read, sorted, stored, and then joined. The one data structure that does not improve the join operation is the indexed data structure. This structure, due to the random nature of the retrievals, increases the time to perform the join versus the join using an algorithm that uses unordered-unindexed data.

The key to the join is to either compare each tuple of one relation with each tuple of the other relation or to use some method of sorting or grouping to reduce the comparison range. The reduction of the volume to be compared is critical because this operation is an NP-complete operation, and by reducing the volume of the input significant decreases in performance time can be achieved. However, to fully use grouping to improve the performance of the join, multiple processors are required to facilitate the joining of the groups of tuples from the relations. Figure 26 presents the performance graph using a single processor with multiple disks to reduce the I/O time. Compare this with the performance shown in Figure 27 that uses multiple processors and the same join algorithms. The performance of the multiple processors case is improved but the times are still very slow. Therefore, Figure 28 uses a hashing technique to create disjoint groups of tuples from each relation, where the corresponding groups or buckets of tuples have the same hash value for the join attributes and potentially have the same actual value for the join attribute(s). This changes the join from joining relation R with relation S to joining  $R_1$  with  $S_1$ ,  $R_2$

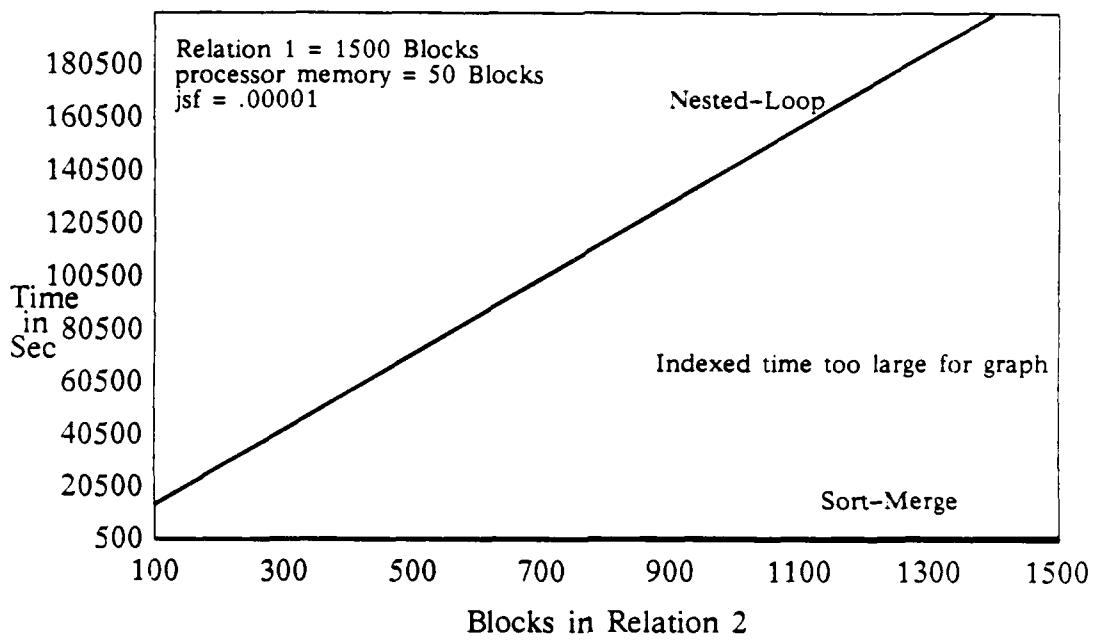


Figure 26. Join Performance using a Single Processor and Multiple Disk.

with  $S_2$ , and so forth. If there are  $p$  processors available to perform the join, then joining the entire relations requires each processor to compare  $(R \times S)/p$  blocks to complete the join. Using the bucket method, each processor compares  $(R/p) \times (S/p)$  blocks or combining terms,  $(R \times S)/p^2$  blocks to complete the join. This shows that creating the series of buckets to be joined significantly reduces the workload of each join processor.

Figure 28 shows the significant performance increase that may be achieved by grouping the appropriate tuples to be joined. Figure 28 presents one situation of the join using the improved techniques of using a bucket join. However, the result presented have used some assumptions that may impact the performance. The first and most significant assumption of the bucket join is that the buckets formed by the hashing function are all approximately equal. If the hash function does not provide the ability to provide equal size buckets, the time required for the bucket join may significantly increase. Therefore, the next section describes the limitations and features of the bucket join.

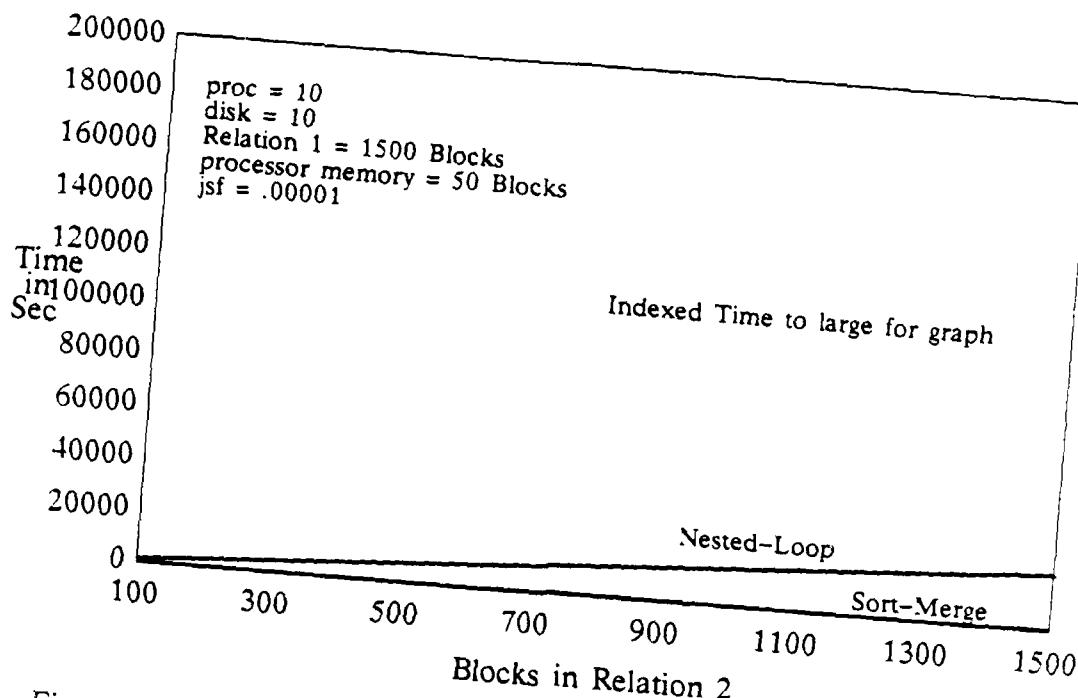


Figure 27. Join Performance using Multiple Disks and Multiple Processors.

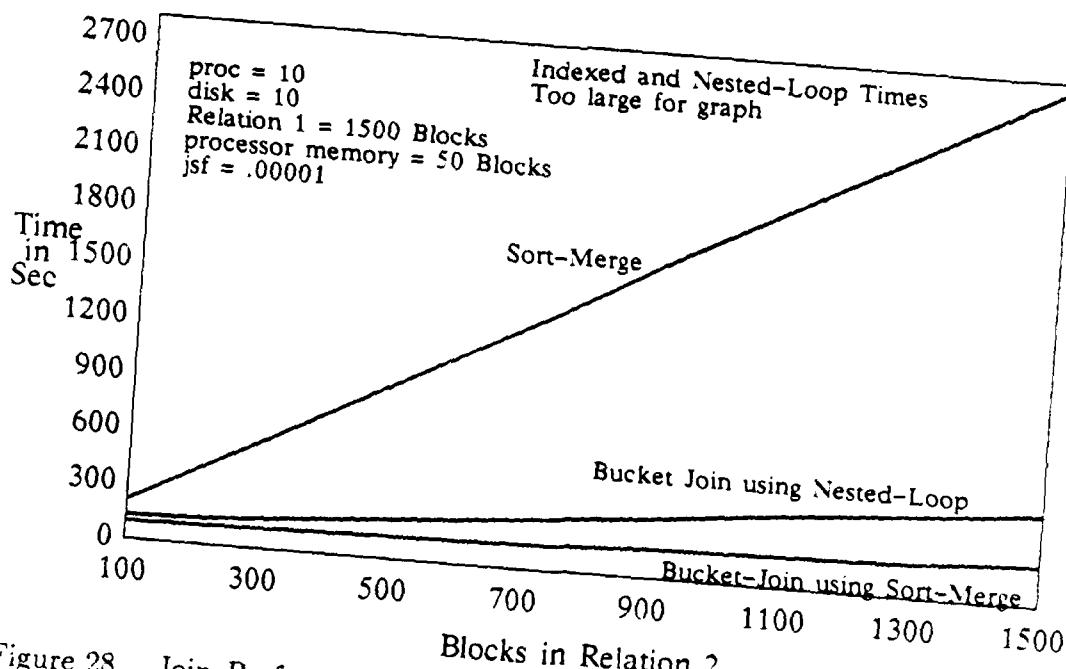


Figure 28. Join Performance, including the Bucket Join, using Multiple Disks and Multiple Processors.

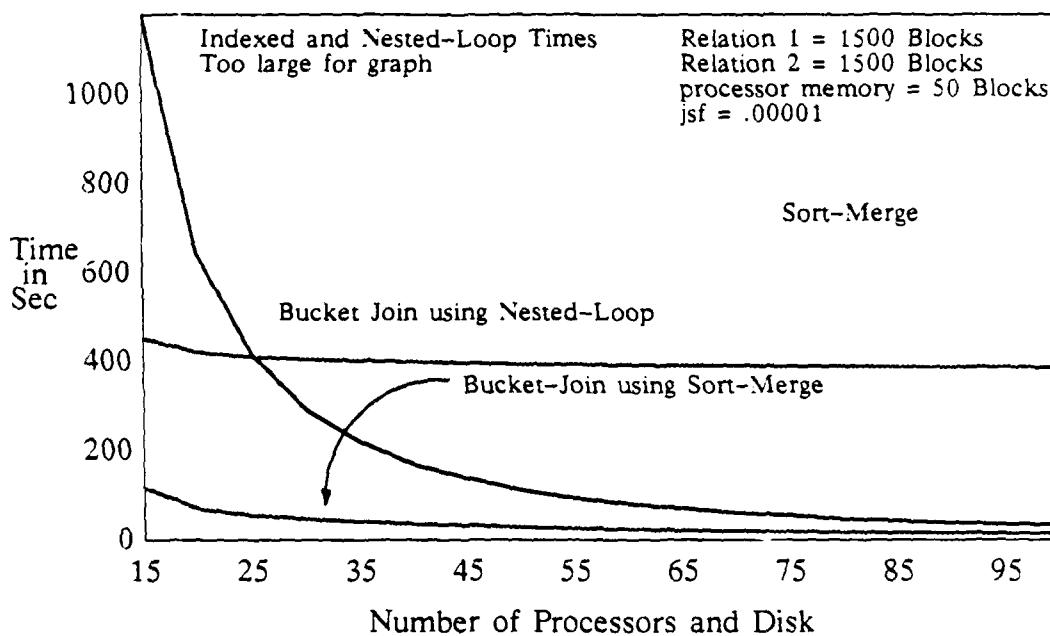


Figure 29. Join Performance Effects of Increasing Processors and Secondary Storage Using Different Algorithms

#### 7.4 Bucket Join

The join environment is critical to the performance of the operation. The environment includes the number of processors; the amount, speed, and availability of secondary storage; the amount of memory available in each processor; and the performance of the communication between processors. The environment affects the performance of all join algorithms. However, the bucket join is more sensitive to some of the environment factors than the more traditional join algorithms, sort-merge and nested-loop. Figure 29 illustrates the performance gain of increasing the number of processors available for the processing of the join.

The bucket type join [22,24,26,48,80] utilizes a two phase algorithm. The first phase is the hashing of the tuples to form the buckets and transmitting the buckets of each relation to the appropriate processors. The second phase of the bucket join is the application of one of the join algorithms to the buckets of each relation at

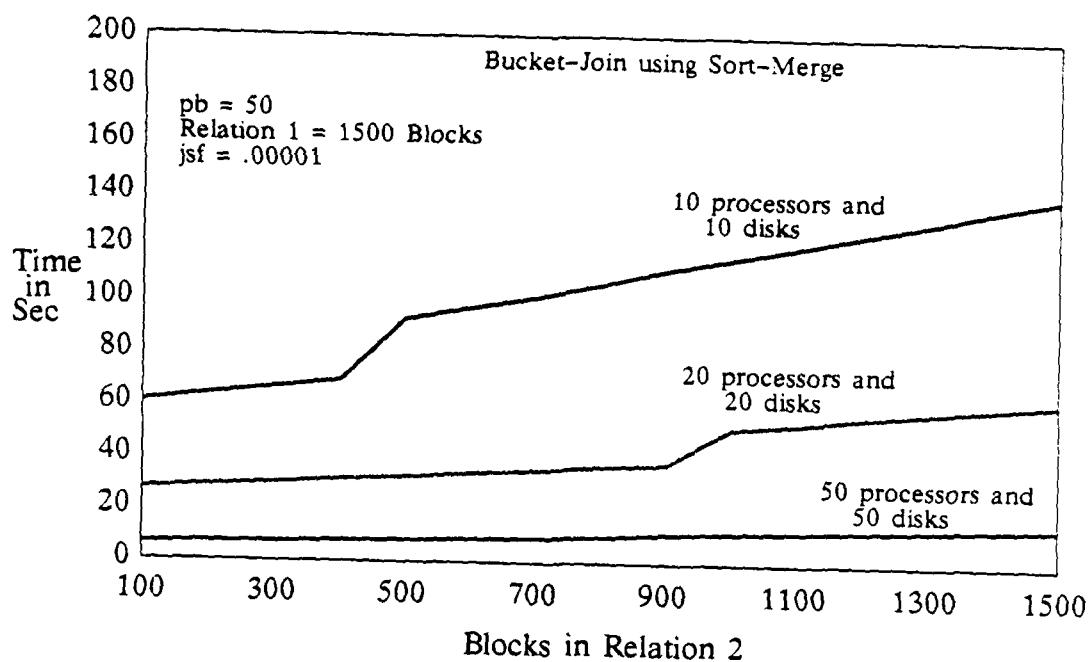


Figure 30. Join Performance Effects of Increasing Processors and Secondary Storage Varying the Input Size

each processor. Therefore, the bucket join still uses the same algorithms as the traditional join techniques except that the amount of input is reduced. The results, shown in Figures 29 and 30, indicate the significant decrease in performance time by the corresponding increase in processors (assuming a corresponding increase in secondary storage capability). But, the performance improvement by adding more processors has to be limited by some bound of efficient usage of the processors and also bounded by the effectiveness of the hash functions in equally dividing the buckets.

The efficiency of the bucket join depends upon the ability to form equal buckets of data to be joined. Therefore, the best case for this join operation is when each bucket formed is equal in size. The worst case is when one of the buckets would contain the entire relation. If the worst case condition existed for one relation but the other relation formed equally distributed buckets, the operation would approximate the number of blocks to be compared using more traditional join techniques such

as the nest loop ( $R \times (S/p)$  or  $(R \times S)/p$  blocks to be compared). To improve this situation, the hashing can be applied recursively to more equally distribute the tuples of each relation. This provides the capability of even the worst case to be handled in a reasonable manner.

The second phase of the bucket join actually performs the join. The methods modeled here to perform the join are nested-loop and sort-merge. If the bucket sizes are much smaller than the processor memory, then a hash join, based upon a hash of the tuples within a bucket to a memory location, can be used which may provide very good performance [26]. However, the concern here is how to handle all cases which includes cases that exceed the capacity of the processor memory. In these cases, the processors that exceed their memory capability could hash the buckets they receive passing buckets onto further processors. This recursive application of the hashing allows the size of the buckets to be regulated if there are enough processors to allow this. Obviously, there are cases when the total size of the relations exceeds the total processor memory size. This requires the processors to use some form of secondary storage to store tuples and to allow intermediate data to be stored during the join processing. Therefore, the speed of access of secondary storage could influence the performance of the join operation. Figure 31 shows that even when secondary storage is used, fast secondary storage versus slower disk secondary storage does not provide a significant performance increase.

The next element of the environment is the communication capability of the processors. It is assumed that the processors have the ability to pass data from one processor to any other processor. This does not mean that there needs to be a direct path from each processor to each other processor, but that there exists a capability for data to be routed through intermediate processors to reach the desired destination processor. One advantage of the bucket join is that only at the beginning of the process when the relations are being hashed does information have to be shared among the processors. After the individual processor receives all of its data, it does

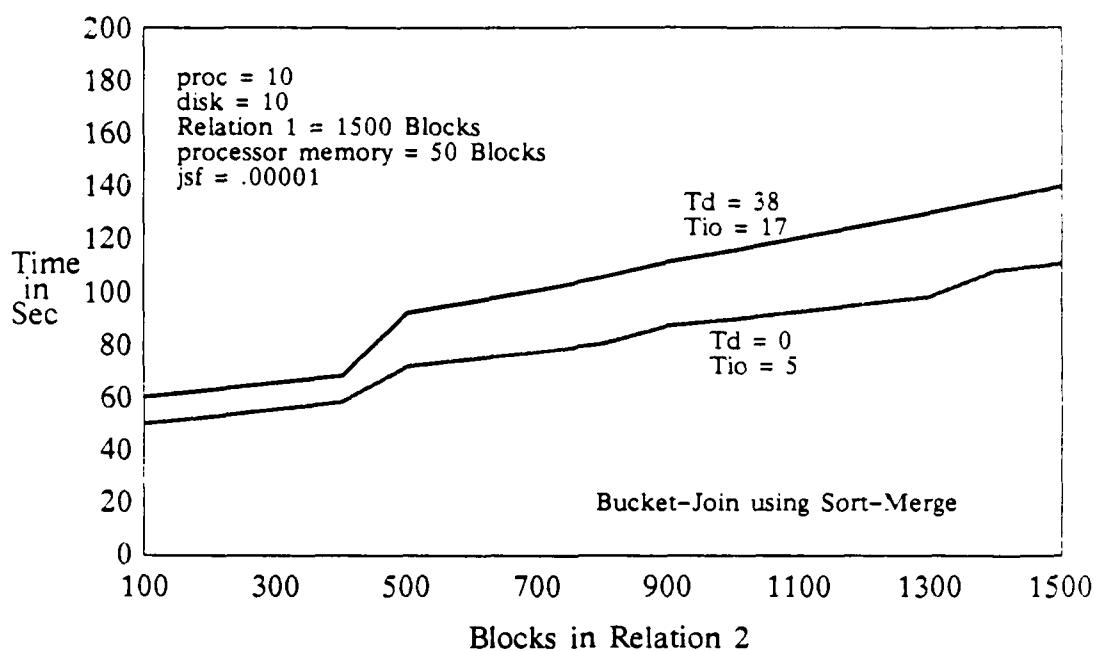


Figure 31. Join Performance Effects of Decreasing Access Speed of Secondary Storage.

not have to communicate with any other processors to complete its portion of the join. This is different from the nested-loop algorithm that must continually pass additional data to be joined and the sort-merge algorithm which requires several transfers of portions of the relations to complete the sorting of the relations. Since all of the join algorithms require some inter-processor communication, Figure 32 shows the effect of improving the speed of inter-processor communication. Included in the effect of inter-processor communication time is the time required to pass the results to some form of back-end. Currently, the back-end is modeled to be a data sink that does not create any contention within the inter-processor communication. Therefore, the volume (as determined by the join selectivity factor) of the results directly effects the time to complete the join. Figure 33 illustrates the effect of join selectivity factor in the performance time of the join.

The last factor of the performance time of the join is the number of processors available to perform the join. The number of processors and the amount of memory

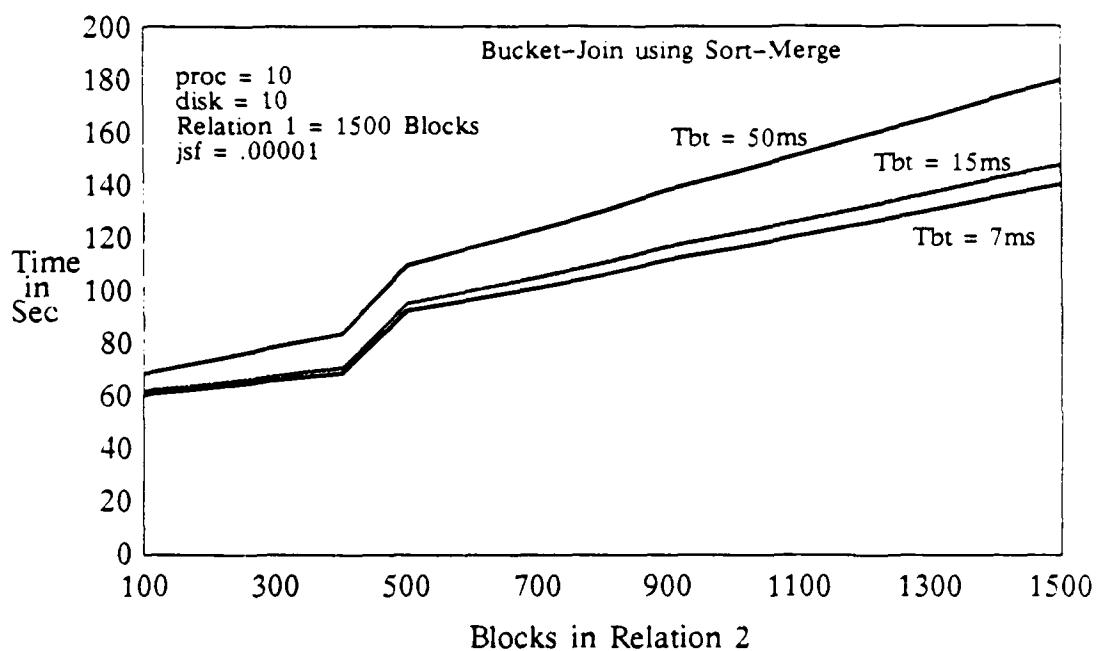


Figure 32. Join Performance Effects of Communication Speed.

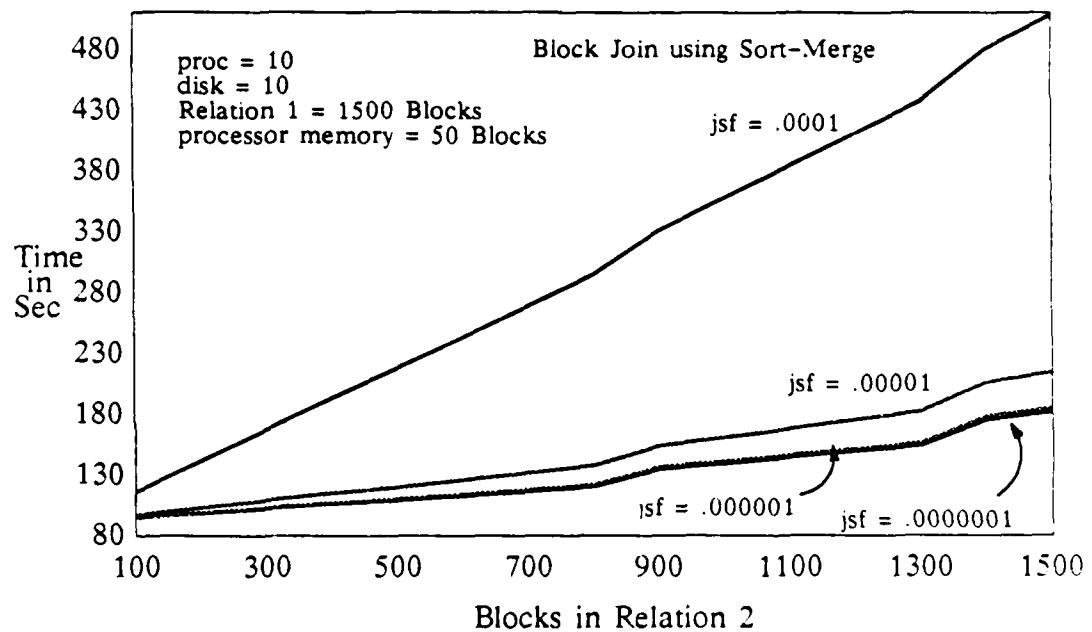


Figure 33. Join Performance Effects of Join Selectivity Factor.

of each processor are combined to describe the capability of the processors in this discussion. The goal when processing the join is to have enough processor capability to contain all of both relations in processor memory. If the processor capability is generated by very large processor memory, the processing times are slower, due to the larger workload for each processor. Therefore, the ideal goal for the number of processors would be to have  $R \times S$  processors if using the nested loop algorithm. This allows each processor to compare the smallest portion of each relation that is feasible. If using the sort-merge algorithm,  $2R+2S$  processors would be the optimal number of processors. This would allow  $R+S$  processors to each sort one block of tuples and leave  $R+S$  processors to form binary trees to merge the sorted blocks plus merge the final sorted relations. The bucket join, due to the grouping of tuples by value, would need  $R$  processors, assuming  $R > S$ . This allows each processor to process one block of the largest relation and a portion of a block of the smaller relation. Thus, the block join requires the least number of processors to provide the optimal processing environment.

The bucket join achieves a more optimal processor environment than any of the other algorithms. Even this environment may not be a realistic number of processors to have in the system. Figure 30 showed the effect of increasing the processors but it does not show what the effect of increasing the processor memory size, to increase the processor capability, will have on the performance of the operation. Since increasing the memory size would reduce the amount of information that would have to be stored in secondary storage, the performance time should improve. However, by increasing the memory size, the processing time increases because the amount of processing to be done by the processor, sorting or comparison of blocks, increases. Therefore, the increasing of processor memory improves the performance but not as significantly as might be expected. Figure 34 shows the effect of varying the memory size of each processor performing the bucket join.

The bucket join is the best join for all equi-join retrievals. This was shown

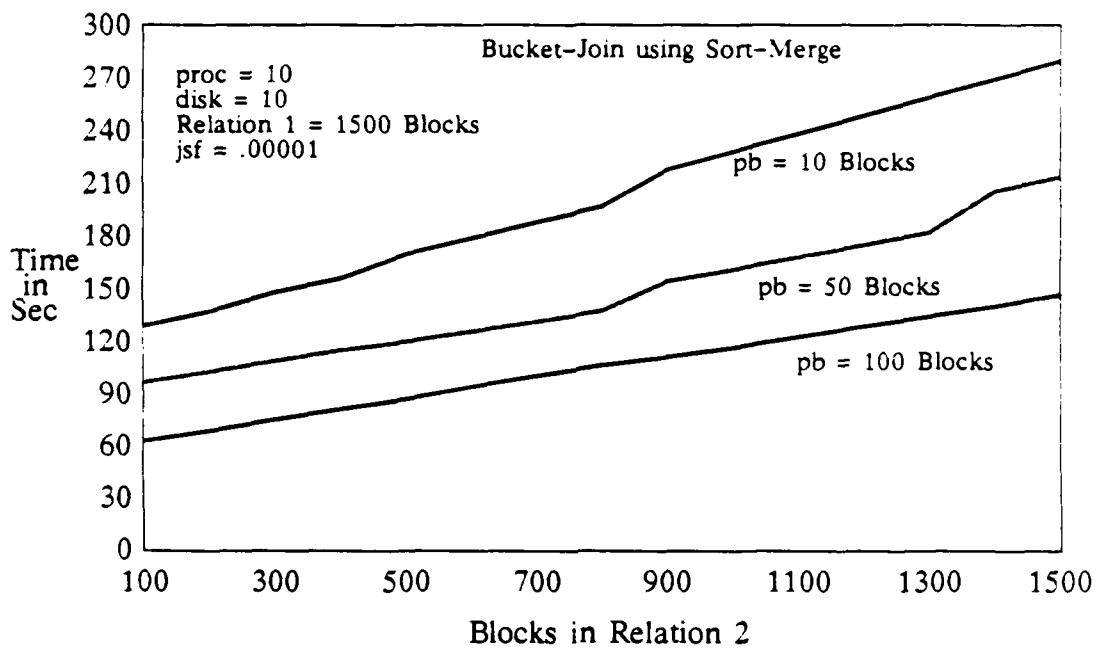


Figure 34. Join Performance Effects of Increased Processor Memory.

above. The sort-merge algorithm within the bucket join provides the best performance results except when each processor contains one or two blocks of each relation. Then the nested-loop algorithm performs better. However, one more consideration in selecting an algorithm is whether the time to produce the first results or the time to produce the entire set of results being more important. If the time to first result is the desired performance element, the nested-loop algorithm will produce results much sooner than the sort-merge algorithm because it starts actual comparison of the two blocks of tuples immediately. The sort-merge algorithm has to sort the blocks of each relation before actual comparison of the relations begins. However, the sort-merge will complete the entire operation sooner than the nested-loop algorithm. The focus of the investigation here has been on the performance time of the complete operation. So, the bucket join using the sort-merge operation is the best algorithm, no matter what the input data structure of the relations, for performing the equi-join.

## 7.5 Update

The update operation is the only operation considered that does not retrieve data in response to a query. The update operations add, modify, or delete tuples from existing relations in the database. All of the update operations require two phases: a determination phase as to the correctness of the update and the actual modification of data within the database. Due to the determination of correctness phase, what seems to be the best data structure for updates may not be as efficient as other data structures. The following sections compare the performance time, which is dominated by disk accessing, necessary to perform updates using the various data structures.

*7.5.1 Insertion* The insertion operation adds new information into an existing database. The first element of the performance time of the insertion is the level of integrity checking enforced within the system. If no integrity checking is done, such as for a history type database that allows duplicate entries, the fastest insertion would be one that could place the data at any available space. This type insertion would require one block to be read, the tuple added to the block, and the block written back to disk. This provides the best possible performance to complete the insertion. However, the performance models developed previously require a minimum level of integrity checking. The result of this is that the insertion, with the unstructured environment described above, requires the entire relation to be read and scanned to perform the integrity check. Thus, the insertion time is increased from two disk accesses to  $R + 2$  disk accesses (where  $R$  is the number of blocks in the relation). The above example illustrates the major element in completing the insertion operation—disk accesses. Next, the performance of inserting a tuple given the different data structures in the multiple processor–multiple disk environment will be examined.

The multiple disk environment allows the overlapping disk accesses. This im-

proves the performance of an operation that requires reading several blocks of a relation. Using this concept, the time to perform an insertion with the unordered-unindexed data structure, with  $R$  blocks in the relation and  $d$  disks, requires  $R/d + 1$  disk reads/writes. However, the indexed data structure requires the index to be updated and the insert can occur in any block with extra space. By updating the index, the integrity checking is also accomplished. Thus, the insertion requires approximately  $L + 2$  disk I/Os, where  $L$  is the number of levels in the index. Since the fanout ratio of the index is user controlled, the maximum number of levels in the index should not exceed 5 or 6. This means the insertion with the indexed data structure requires approximately 7 or 8 disk reads or writes. The contrast in data structures illustrates that the indexed insertion requires a constant number of disk I/Os to perform the insertion, while the disk I/Os using the unordered-unindexed data structure increases with the relation size.

The insertion of a tuple in the ordered data structure is similar to the unordered case described above. The difference with the ordered data structure is that the entire relation does not have to be read in each case. The expected value of disk reads to find the insertion point for the new tuple is half of the relation. The insertion in the ordered relation also requires the ordering of the relation to be maintained, though. This means the tuples within blocks may have to be moved to make room for the insertion. Therefore, the ordered insertion depends heavily upon the expected values of finding the appropriate insertion location and number of blocks that need to be reorganized. The implication of this is that the entire relation will be read to find the proper location or will be read and written during the reorganization. However, using multiple disks, the number of disk reads or writes is reduced to the portion of the relation stored on a single disk. This, like the unordered-unindexed data structure, varies with the size of the relation.

The final data structure is the ordered-indexed data structure. An insertion with this data structure requires the constant number of disk I/Os to update the

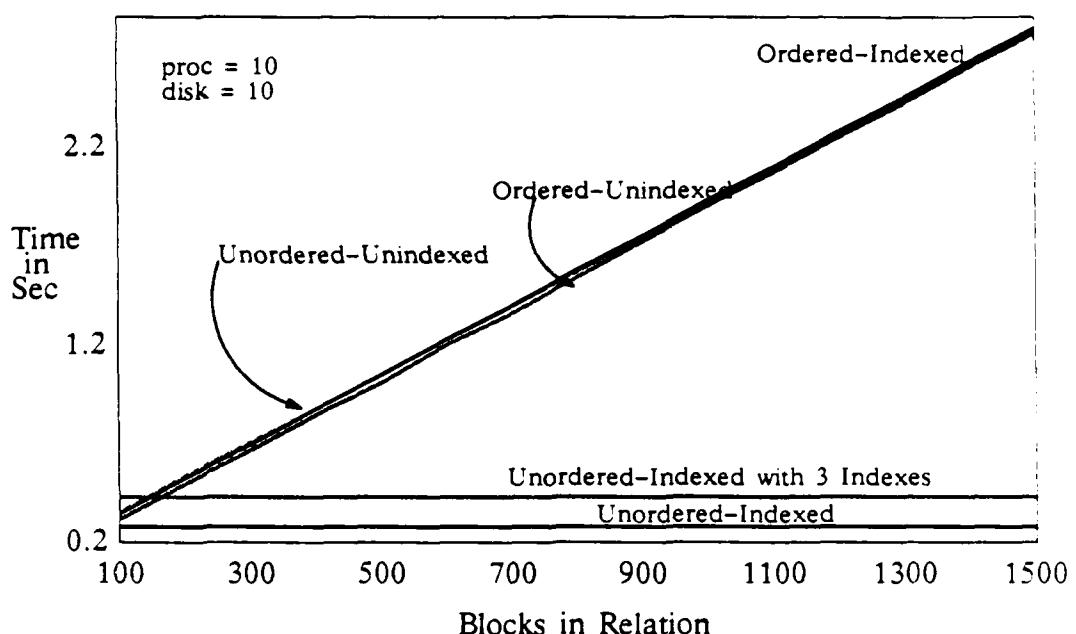


Figure 35. The Performance of Inserting a Tuple

index and to find the proper insertion location. Then the remainder of the portion of the relation stored on the disk following the insertion point would be reorganized to make room for the insertion. This may require several disk accesses or just a few accesses but it is assumed for modeling purposes half of the blocks contained on the disk will be reorganized.

The results presented above show that the indexed case presents the best structure for completing an insertion in an on-line situation. The one effect not discussed is the time required to maintained multiple indices. However, each index will require another  $L + 2$  disk I/Os to be updated. If the indices are contained on separate disks and several processors and disks can be used to update the indices the performance time would be the same as updating a single disk. Figure 35 illustrates the various insertions with the different data structures, showing the excellent performance of the indexed data structure. The performance times presented also show an indexed case that requires one processor to update two additional indices and its effect upon the performance time.

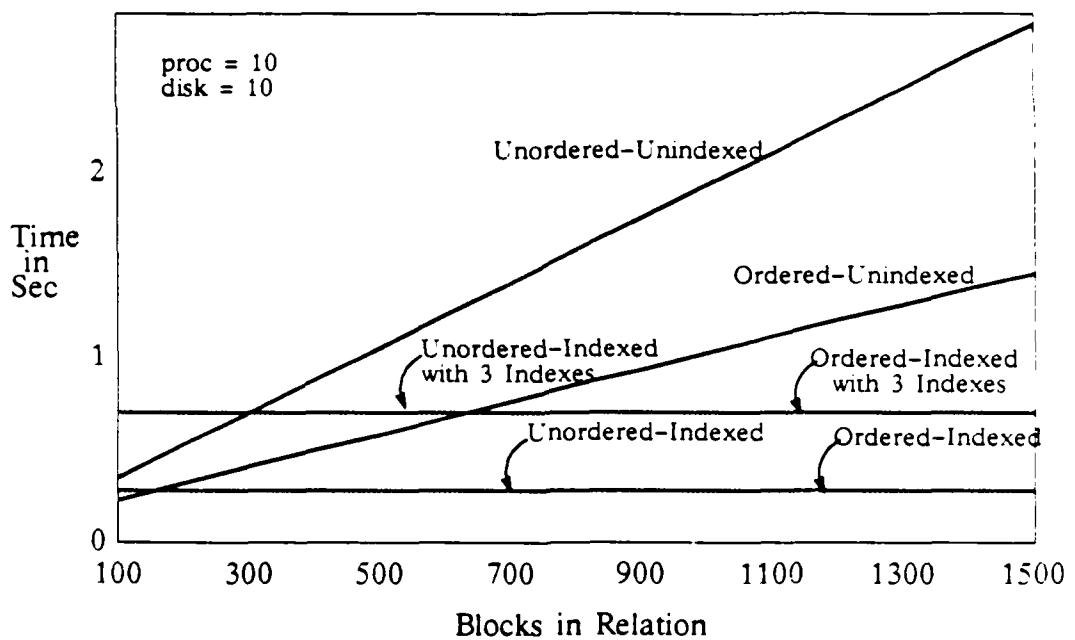


Figure 36. The Performance of Deleting a Tuple

**7.5.2 Deletion** The deletion of a tuple is similar to the insertion except no blocks will need to be reorganized to maintain the proper ordering. Therefore all the data structures provide performance similar to the insertion performance. However, the ordered case provides an improvement over the unordered case for all situations. The indexed cases still provide a constant time to perform the operation no matter what the size of the relation, thus providing the best performance. Figure 36 shows the performance times of the different data structures to complete a deletion of a tuple.

**7.5.3 Modification** The final update action is a modification of a tuple. Actually the modification is considered to be a deletion and an insertion combined. Therefore, the performance results are similar to the insertion and deletion. Figure 37 shows the performance results of performing a tuple modification.

The results presented for the update actions show that the best data structure for updates is the unordered-indexed data structure. This says that if the user

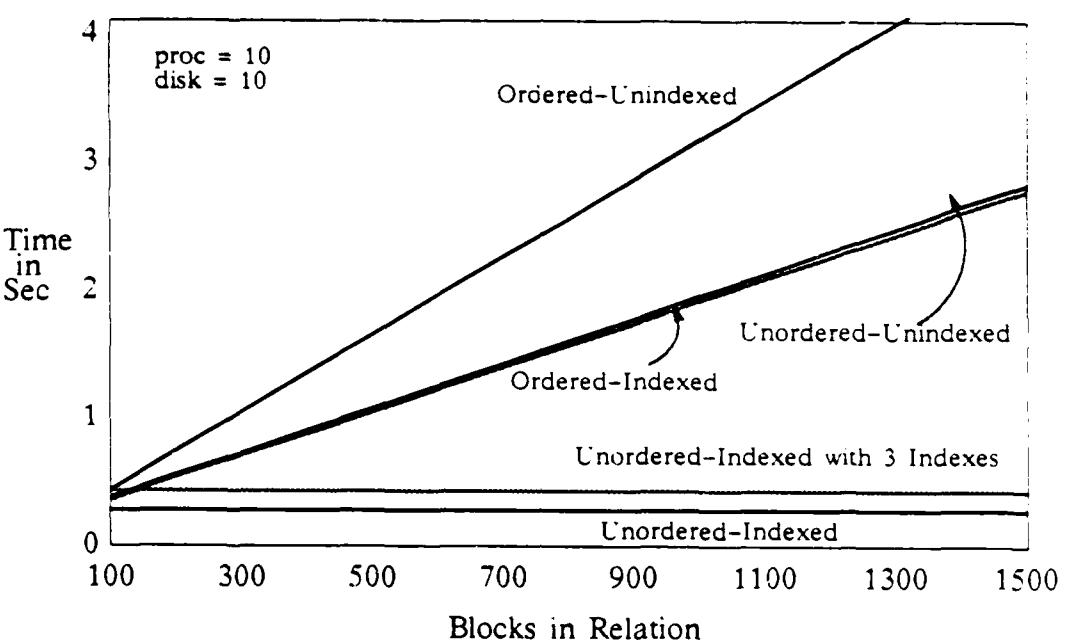


Figure 37. The Performance of Modifying a Tuple

requires many on-line time-critical updates, the best way to store the relation is in an unordered manner with an index for the key of the relation.

#### 7.6 Performance Conclusions

The performance results presented in this chapter do not provide a "best data structure" for storing relations; however, some conclusions about the "better data structure" for given situations can be drawn. The data structure has no impact on the performance of the project. Also, the join processing that provides the best results (bucket join) does not depend upon a data structure because the first step scans the relation and redistributes the relation. Therefore, any of the data structures will provide the best environment for performing the project or join operations. Thus, the only operations that depend upon the data structure are the select and update.

The select operation performance results presented two conflicting views. The first view shows that if the main retrievals from the database consist of queries that produce results consisting of a single tuple or a few tuples, the best data structure

is indexed, either unordered-indexed or ordered-indexed. However, when the results produced in response to a query consist of a group of tuples, the unordered-indexed case provides the poorest performance because it requires random retrievals of tuples. The best performance is provided by the ordered-indexed data structure, with the ordered-unindexed and unordered-unindexed providing similar results. The advantage of ordered-indexed data structure is the direct pointer to the beginning of the tuples that satisfy the query. Thus, the select performance presents conflicting views of the "best data structure". Therefore, the results of the select data structure must be balanced with the results of the update operations.

The update operation performances presented all assume the update must be done immediately. If the update does not need to be done immediately and the updates can be accumulated and done together in a batch, the update operation then becomes more of a select-type operation which favors the ordered structures or unordered-unindexed structure. Therefore, the individual database user requirements must be defined to allow the best data structure. However, if the requirements of the database are mixed or are not definable, the probable data structure mix is the unordered-unindexed data structure, because the unordered-unindexed data structure provides middle of the road performance for all situations and the performance of the unordered-unindexed data structures can be linearly increased by increasing the number of processors and disks until each disk contains a single block of the relation. In this optimum case, the select or update operations would require each disk to retrieve its one block, thus requiring the time for only one disk access.

The final operations are the binary operations. The results presented showed that the join operation is best performed by a distributed environment that groups the information independent of the stored data structure. The product operation ignores any data structure in its processing. Therefore, the data structure does not impact the binary operation processing in a multiple processor environment. The consideration of the performance of the binary operations is the ability to access

the relations in parallel, to provide the multiple processors with data as quickly as possible. This implies that the data structure consideration of these operations is that the relations are stored in the round-robin fashion on the disks to allow parallel access to the relations.

The conclusion is that the main requirement for all processes is parallel access to the relations for over all best performance in all cases. When the data structure is being optimized for single retrieval or update operation, the relation should still be stored distributed on the disks to provide the parallel access of the relation when the optimized case does not fit the retrieval. Table 4 summarizes the effects of the environment parameters on the performance of the various operators. Therefore, for the case of select-FT case the table indicates that indexing is desired. However, if the relation data structure is indexed, the tuples should still be stored on several disks. This will not improve the select of a single tuple from the relation but provides better performance when the relation is needed for a different operation.

Table 4 summarizes the best performance for the given set of time parameters used for this evaluation (e.g.,  $T_{io}$ ,  $T_b$ ,  $T_{bt}$ , ...), assuming a multiprocessor environment. This does not mean that the performance of the "best" column will always provide the "best" performance for all combinations of performance parameters and architectural constraints. The performance summary presented shows how the query processor designer would use the models to produce a set of similar "best" results for their actual case to guide the design process. The general column of the summary table provides an overview of what data structure is "always" optimal if no data structure currently exists. This means that for the select it is not feasible to have to sort the relation first to provide better performance for a single retrieval. Thus, Table 4 illustrates the use of the analytical models as a building block in the structured approach to designing a database query processor.

**Data Structure or Algorithm  
for Best Performance**

	Best Case	General Case
Select - Single Tuple Result	Indexed-Unordered	Unindexed-Unordered
Select - Multiple Tuple Result	Indexed-Ordered	Unindexed-Unordered
Project	N/A	N/A
Insert	Indexed-Unordered	Unindexed-Unordered
Delete	Indexed-Unordered	Unindexed-Unordered
Modify	Indexed-Unordered	Unindexed-Unordered
Join	Bucket-Join	Undetermined
Product	Nested-Loop	Nested-Loop

Table 4. Summary of Performance Results

## *VIII. Multi-Step Query Performance*

The typical retrieval from a database requires more than one relational operation to complete. All of the focus on performance to this point has been on individual retrieval operations. Next, the effects of combining the individual retrieval operations are explored. First a general form of expressing a multi-step query—a query tree, is explored. Then the individual query steps are examined to determine how they fit the general form of the query and how multi-step queries may be executed.

### *8.1 Query Tree*

The relational database exists to provide data in response to user queries for data. The queries may require several steps to provide the response in the user-defined format. The series of steps necessary to complete the query correspond to the relational operators and are often expressed in a tree form. This tree representation of the query steps is called a query tree [19]. Figure 38 illustrates a query tree.

The query tree is an operator tree that shows an internal representation of the steps necessary to complete a query. By using a query tree, any relational algebra equivalent query may be represented since the query tree shows the relational algebra operators necessary to complete the query. The internal nodes of the query trees describe the relational operators necessary to answer the query. The leaves of the tree describe the relations used by the relational operators. The query tree also shows the execution order of the relational operators. It is assumed that any optimization done by manipulating the execution order of the operators [73,79] has already been completed.

The query tree is actually a parse tree showing the relational operators and their execution order [79]. By definition, a tree is a finite set of one or more nodes such that: (1) there is a specially designated node called the root; (2) the remaining nodes are partitioned into  $n$  disjoint sets,  $T_1, T_2, \dots, T_n$ , where each of these sets is a tree.

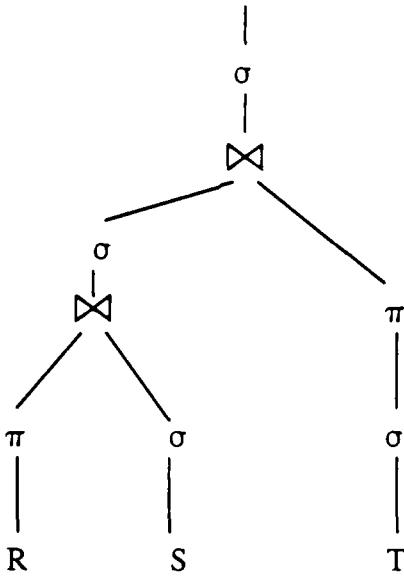


Figure 38. A Simple Query Tree

$T_1, T_2, \dots, T_n$  are called subtrees of the root [39]. Also, the query tree is a directed graph in which the nodes represent precedence relations between tasks. Figure 38 shows a typical query tree and Figure 39 shows the expanded directed graph form of the query tree. These properties make the query tree compatible with a computation graph [45,55]. Cesarini uses the compatibility with a computation graph to construct query execution graphs to explicitly show the independent parallelism possible for a given query tree [16].

Parallelism is the use of multiple processors to effectively work on a problem concurrently. Independent parallelism, intranode parallelism, and pipelining are three forms of parallelism. In independent parallelism, the tasks do not share data or depend on the results of a task that is currently executing, allowing the tasks to be processed concurrently or in parallel. Intranode parallelism, as seen in previous chapters, is parallelism allowed by the specific algorithm and data structure used to complete a task. In intranode parallelism, the algorithm must effectively allow different processors to work on the same or shared portion of the task without corrupting

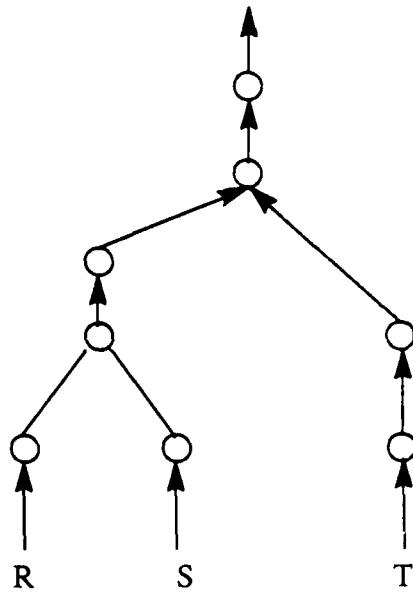


Figure 39. Directed Graph in the Query Tree

the data (this type of parallelism is not possible in all cases). Pipelining implements multiprocessing by having data-dependent processes cooperate in creating a data pipeline. The pipeline passes partial data from a process to the next process allowing several dependent processes to be active concurrently by using partial results from the previous process. Therefore, parallelism improves performance by the use of processor allocation.

Parallelism is one method of improving the performance of completing the processing to satisfy a query. The query tree provides other opportunities for possible performance improvements. One method of performance improvement, using the query tree, is query optimization. Query optimization uses the mathematical basis of the relational operators [18] to reconfigure the order of the operators in the query tree [79]. This manipulation of the order of the operators to make the query structure more efficient bases its manipulation on the amount of data being passed from node to node in the query tree and tries to reduce the data as soon as possible [73,79]. Since the performance of the query depends upon the amount of data to be processed.

it is useful to annotate the query tree to describe the amount of data used at each step of the processing.

Each of the operations in the query tree produces a relation as its results. The arcs of the query tree represent the passing of relations from node to node. Therefore, there is a size or amount of data associated with each arc, which is dependent on the previous operation. Using the query tree of Figure 40, assume the sizes of relations  $R_1$  and  $R_2$  are  $n_1$  and  $n_2$ , respectively. Then arcs  $a_1$  and  $a_2$  would have sizes  $n_1$  and  $n_2$ . The size associated with arc  $a_3$  is a function of operation  $x_1$  and its input  $a_1$ . Thus, the size of  $a_3$  is  $f_1 n_1$ . Correspondingly, the size of the relation associated with  $a_4$  is  $f_2 n_2$ . Arc  $a_5$  is the result of binary operation  $x_3$ . Therefore, the size of  $a_5$  depends upon the size of both inputs  $a_3$  and  $a_4$ , and the operation  $x_3$ . The size of the results of  $x_3$ , arc  $a_5$ , is represented by  $f_3(f_1 n_1 \times f_2 n_2)$  (the product combination of inputs is normally considered for the join and product operations and  $f_3(f_1 n_1 + f_2 n_2)$  for other binary operations, although by reducing the selectivity factor the product representation can be applied). The range of  $f_i$  is 0 to 1, providing a range for the output from 0 to  $(f_1 n_1 \times f_2 n_2)$ . Since  $f_1$ ,  $f_2$ , and  $f_3$  control the volume of the output, they are called selectivity factors.

The selectivity factor is a critical element because it determines the size of the resulting relation which may be the input into another operation. The effect of the selectivity factor is greater than it may seem. A simple example will be presented that illustrates the binary operation information explosion possible. Suppose there is binary operation,  $x_n$ , that has inputs,  $R_n$  and  $R_m$ , which each have 150 blocks. If the selectivity factor is 1, the results would consist of 22,500 blocks of information ( $1 \times (150 \times 150)$ ). Previously, it was noted that this method of computing results may not be valid for some binary operations, such as the product or join, because it lacks sensitivity of the combination of the individual tuples within the blocks. Using this concern, further examination of the example shows that each block of the relations contains 100 tuples. The product now actually becomes.  $(100 \times 150) \times (100 \times 150)$ .

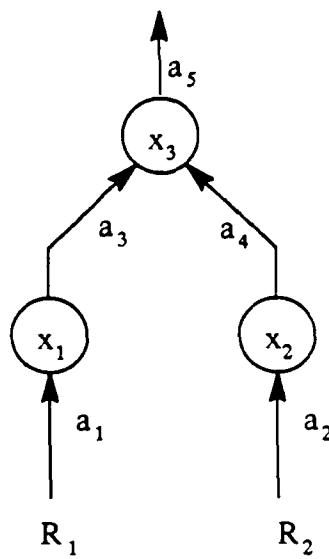


Figure 40. Example Query

plus the result for the product or join is a combination of the two input tuples, meaning that only 50 tuples will fit in a block for the results. Therefore, with selectivity factor of 1, the operation produces 225,000,000 tuples. With 50 tuples fitting in a block, there are 4,500,000 blocks produced. This illustrates the dramatic information explosion possible with binary operations.

The concept of the query tree is that the operations take many inputs and reduce them to a single solution. This is true, but the shape of the query tree seems to imply that the data volume also is reduced as it proceeds up the tree. However, as illustrated above this may not be true; instead, the resulting relation may be much larger than the sum of the inputs. This has serious implications in the time to complete the query because the size of the input is the main element in the time to perform an operation and one of the main time requirements of the operation is the time to store and retrieve the intermediate results from secondary storage. Therefore, reducing the amount of intermediate relation storage is a critical element in optimizing the performance of executing the query. The next section looks at how

to reduce the intermediate storage of relations.

## 8.2 Combined Operators

The purpose of combining operators is to reduce the need for passing and storing intermediate relations by performing as many operations as possible while the data is contained in the processor memory. There are two types of relational operators: unary and binary operations. The unary operators, select and project, are data reduction operations. Select and project reduce the input by either selecting horizontal or vertical portions of the relation. Thus, if the query requires a select followed by a project, there is no reason that these operations cannot be combined to be performed together. The operation performs the select, producing tuples that satisfy the selection criteria, and then the results can be reduced by only producing the attributes that are necessary to satisfy the projection.

*8.2.1 The Sel-Proj Operation.* The sel-proj operation selects the tuples that satisfy the selection criteria and the projection is applied to these tuples. The effect is that the projection operation is applied only to that portion of the relation already in memory, reducing the I/O time required. If the sel-proj operation is required for a retrieval that requires only a project, the sel-proj operates as a projection operation, selecting all tuples, while if only a selection operation is needed, the results of the select are not altered. The combination of the two operations reduces the intermediate reading and writing of temporary relations, improving the performance of multi-step queries.

Figure 41 shows the performance of the sel-proj operation (using the unindexed-unordered data structure) versus the sum of the time to perform the separate select and projection operations in sequence. This shows that the reduction of I/O by eliminating the intermediate storage of results in the combined operation does improve the performance compared to performing the two operations separately. Next, the

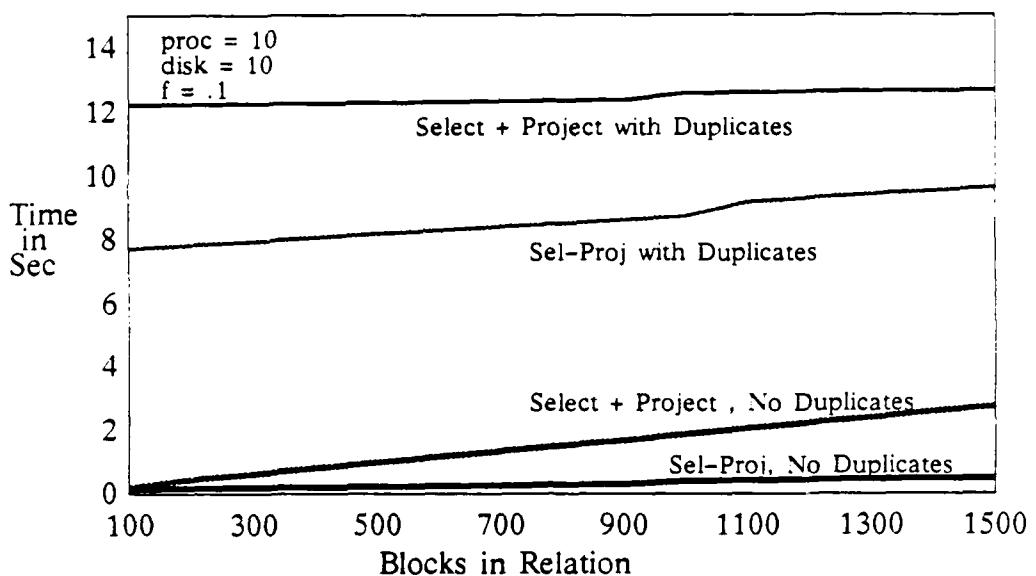


Figure 41. Sel-Proj Operation versus Select and Project using Multiple Processors-Multiple Disks.

possibility for combining other operators is explored.

*8.2.2 Other Combined Operations.* The concept of combining operators is really a simple form of pipelining. Pipelining requires that the operation can operate on a portion of the input without corrupting the final results. Corrupting the results also means not providing the complete set of results. Therefore the only relational operators that satisfy this condition are the unary operators (without special data structures or special constraints). The only unary operators are select and project and it has previously been shown that they can be combined into a single operator. There is however one more possibility for combining relational operators.

The final operator combination combines the sel-proj operator with any binary operator. The sel-proj is an unary operator. It takes a single input relation and produces a single output. The sel-proj can be combined with any binary operator. The binary operations have two input relations and produce a single resultant relation.

Therefore the binary operation can be combined with the sel-proj operation, where the results of the binary operation are reduced by selection and projection before they leave the processor that performed the binary operation. Performing a select and project (with no duplicate removal) on fragments (horizontal fragments) was shown to be valid in Chapter II.

The combination of a binary operator and the sel-proj produces valid results. Next, the performance improvement of combining operators and the limitations of combining operators are discussed to determine the value of combining operators. An example best illustrates the opportunity of improved performance by combining operators. Suppose there is a binary operation that produces 1000 blocks of output followed by a selection and the results of the selection are further reduced by a projection. If the operations are done individually, the intermediate results must be passed from one processor to another or the results must be stored on some form of secondary storage. If the operations are performed as three separate operations, the 1000 blocks of results must be passed to the select operation. The results of the select must be passed to the project. If the select produces 500 blocks of output, then it is necessary to pass 500 more blocks and finally, the project must scan the 500 blocks to produce its results. The "cost" of using three operators instead of combining the three operations is the time to pass 1500 blocks and the time to scan 500 blocks. The consideration is that the scanning of the results of the binary operation does not take any more time in the processor that produced the results as in a separate processor and by performing the operations together, the time required to pass data is reduced to only the final results of three operations instead of one operation.

### 8.3 General Query Tree Form

The combination of the binary operators with the select and project operators provides the opportunity to restructure the query tree to a normal query tree form. The normal query tree form reduces the operations required to complete a query

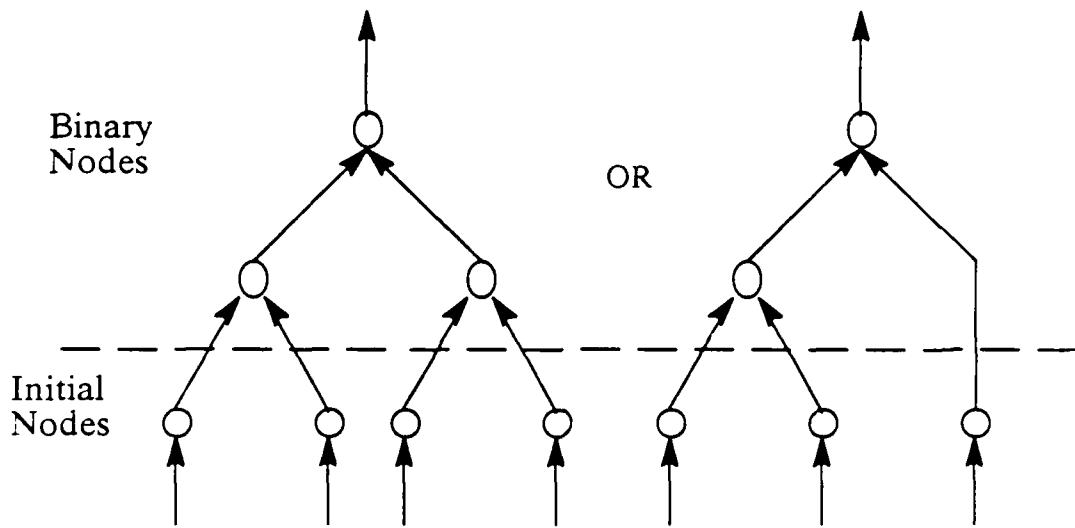


Figure 42. General Normal Form Query Tree

and this reduces the volume of data being transmitted. Figure 42 shows the general normal form query tree. Figure 43 illustrates the query shown previously in Figure 40 in the normal form query.

The normal form query has two types of nodes—the initial nodes and the binary nodes. The initial nodes are the operations that first retrieve the input or base relations. The initial nodes also reduce the base relations as much as possible. The binary nodes consist of a binary operation combined with a select and project that prepares the results of the binary operation for the next binary node, or provides the final formatting of the data for presentation to the user.

*8.3.1 Initial Nodes in Normal Form Query.* The initial nodes retrieve and provide the input to the base level binary nodes. The primary task of the initial node is to retrieve the base relations for further processing. The initial nodes also perform any data reduction that is possible.

The data reduction with a single relation is either a select, project, or both a

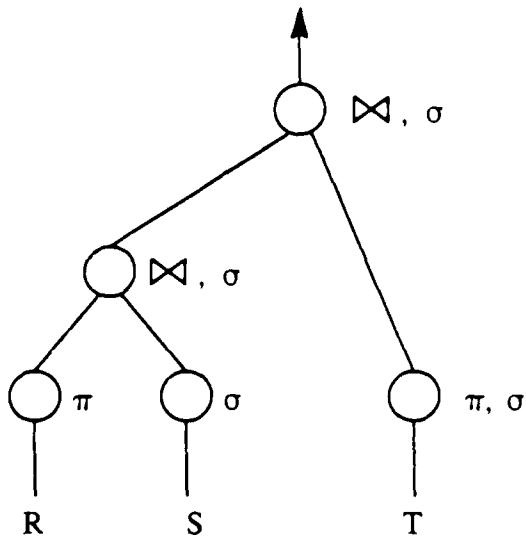


Figure 43. Example Query

select and project, using the sel-proj operator. The initial nodes perform as data filters to reduce the data as it is retrieved from secondary storage. If the project performed by the initial nodes does not remove duplicates, the operations of the initial nodes can be completed by a single scan of the relation as it is retrieved. Therefore, these relations passed to the lowest level binary nodes are ready for the first binary operation. However, these relations may have duplicate tuples. But, the duplicates will be removed by the operation of the binary nodes.

The initial nodes may provide the capability to completely satisfy some queries. If the query calls only for a select and/or a project that does not introduce duplicates, the initial nodes can complete the query.

*8.3.2 Binary Nodes in Normal Form Query.* Each binary node takes two input relations and produces a single output relation. The binary node inputs have been reduced by sel-proj operation of the initial nodes or lower level binary nodes. Also, although the base relations may have been stored as ordered or indexed data,

the data structure entering the binary nodes in general is unindexed-unordered because the key may have been projected out, destroying the ordering and indexing. As described previously, the binary node is a combined operator. It first performs the binary operation, then the sel-proj operator is used to reduce the results to the normal form for the next binary node.

The purpose of the binary nodes is to complete the necessary operations to complete a query. The binary relational operators are product, join, difference, union, intersection, and division. The binary nodes will also be used to remove duplicates (the means of duplicate removal will be discussed later). Next, how the binary nodes implement the binary operations will be discussed.

The operations the binary nodes have to implement fall into two classes – equal comparison operators and compare-all operators. The equal comparison operators are operators that only need to find the tuples that have equal values for specified attribute (or equal tuples where the keys are equal). Thus, if all the tuples with equal attributes can be grouped together, the operator only has to examine each group at a time to complete the operation. The equal comparison operators use groups of tuples of the relations where corresponding groups are formed under the same criteria. For relations R and S, this means that R and S are grouped so  $R_1$  corresponds to  $S_1$ ,  $R_2$  corresponds to  $S_2$ , ..., and  $R_n$  corresponds to  $S_n$ . Besides the equi-join, the other operators that fit this condition are union, difference, and intersection. The most familiar example of the equal comparison class operators is the equi-join. The equi-join may be implemented by sorting each input relation on the join attribute and then performing a merge type comparison. The merge operation works because it needs only the tuples that have equal join attribute(s). Thus, the name equal comparison operator is a descriptive name for the equi-join operator.

The union operator combines two relations that have the same tuple definition. This means that the tuples of both relations have the same attributes defined on the

same domains. The union operator then combines the two relations, eliminating any tuple that may be duplicated. The grouping of the tuples by value insures that duplicate tuples will be in the same bucket or group and the comparison will discover and eliminate the duplicates. The results of the union consists of the combination of the results from each bucket and the results from each bucket consists of all the tuples remaining after the buckets of each relation are sorted and compared by a merge type operation and duplicates eliminated (see description of bucket-join in a previous chapter for complete description of sort-merge processing).

The intersection operator compares two relations and selects only the tuples that appear in both relations. This operation is the opposite of the union. This means that during the comparison phase of the operation only the tuples that are in both relations are selected. By providing groups of tuples formed by the same criteria, the tuples that are equal are guaranteed to be in the same group providing the ability to compare only small groups of tuples instead of the entire relation. Therefore, the intersection is a member of the equal comparison class of operators that may be implemented by the bucket comparison operation.

The final binary operator that may be completed using the equal comparison of the bucket comparison operation is the difference operation. The difference operator compares a first relation with a second relation and removes from the first relation any tuples that are duplicated in the second relation. Again, the operator operates only on tuples that are equal in both relations and the correct results can be provided as long as all the equal tuples are compared. If no duplicate tuples were found the results would then be the entire first relation. Since only the equal tuples need to be found, the bucket-join type processing of grouping the relations into buckets, sorting the buckets, and merging or comparing the ordered buckets, provides the capability to perform the difference.

One requirement of the binary nodes, duplicate removal, does not fit the equal comparison class of operators because it is not a binary operation. However, du-

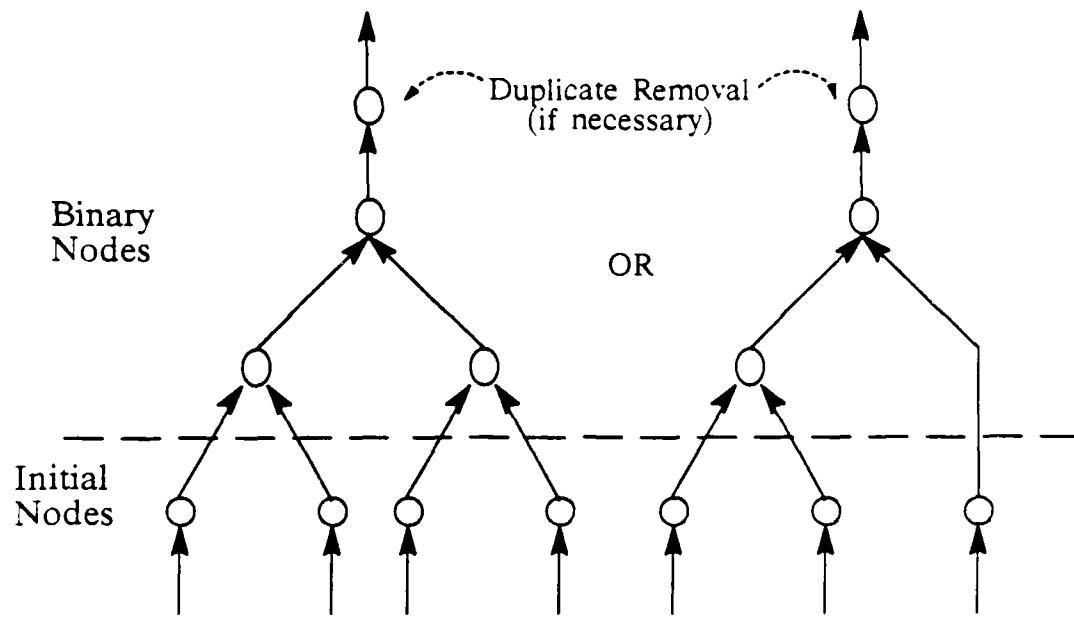


Figure 44. Normal Form Query Tree with Duplicate Removal

plicate removal is a part of the combined operator of the binary nodes when the bucket process with sort-merge is used. Duplicate removal is easily completed by the bucket processing technique because the grouping of tuples and further sorting places duplicate tuples together in the input relations where the duplication is easily identified and duplicates removed. Therefore duplicate removal from the input relations is incorporated in each operation that uses the bucket type processing.

Since the bucket processing eliminates duplicates from the input relations, any duplicates introduced by the projection on the output stage of a binary node will be deleted by the next binary node encountered. If no more binary nodes are required, then a special duplicate removal node needs to be added as the last node of the normal form query tree. Figure 44 shows the modified normal form query tree with duplicate removal. With this extra node to remove duplicates as required and the combined operators, all queries can be mapped into the normal form query tree. However, the binary nodes also may be required to perform compare-all operations.

The binary nodes must also be capable of performing the other binary relational operators—product and non-equality joins. The compare-all operations are operations that must compare each tuple of one input relation with each tuple of the other input relation because just finding the equal tuples will not insure the complete or correct results. These operations require all or parts of the relations to be compared or combined with all or part of the second relation. The product requires each tuple of the first relation to be combined with each tuple of the second relation. The only algorithm that can accomplish this is the nested-loop algorithm.

The nested-loop algorithm compares tuples of each relation block by block to complete the operation. The nested-loop algorithm was discussed as a solution for the equi-join operator in an earlier chapter. The problem of the nested-loop algorithm is the lack of problem reduction capability such as the bucket-join process had. Therefore for two relations with  $n$  and  $m$  blocks, respectively,  $n \times m$  blocks must be compared in the nested-loop algorithm versus the  $(n/b) \times (m/b)$  block comparisons when the processing may be grouped by blocks. However, the nested-loop processing is necessary to complete the product operation.

The join, including the equi-join, is a special case of the product which consists of a product followed by a selection, where the criteria for the selection is referred to as the join condition. The equi-join has a join condition that selects tuples where the join attribute(s) are equal. However, the join condition may also contain a less than or greater than condition. Although the bucket-join does provide a means to group the necessary tuples together, the hash function does not reflect true ordering of tuples by value. Therefore, the nested-loop process is used to perform the join where the join conditions are other than equal conditions.

#### *8.4 Performance of Normal Form Queries*

The individual components of the normal form query tree are the initial nodes and the binary nodes. The characteristics of the nodes of the query tree have been

discussed and the performance characteristics of the individual operations considered in the implementation of the individual nodes. However, the execution of the entire query and its total performance and the use of parallelism in the normal form query tree have not been directly addressed. The following discussion provides this information.

The time to execute a query expressed in the normal query tree form is the time to complete execution of the root binary node plus the time to complete the largest subtree (largest in terms of execution time, where execution time is the time from the first processing of the query until the time of completion the last subtree process, and not necessarily the total time spent executing the subtree). This definition is recursive since each subtree is also a tree. Therefore, the time to execute a subtree is the time to complete the root node of the subtree plus the greater time of its subtrees. By this definition, the time to execute a query in a single processor environment would be the sum of the node execution times. In a multiple processor environment, the possibility of using parallelism may improve the execution time. Therefore, further exploration of the parallelism possible in the normal form of the query tree will be explored.

The three forms of parallelism—*independent parallelism*, *intranode parallelism*, and *pipelining*—all have opportunity for use in the solution of the query. First, *intranode parallelism* is the use of multiple processors in the execution of a single node. Intranode parallelism has already been addressed and shown to be effective for the *sel-proj*, *bucket-join*, and *nested-loop* processing, which are the only types of operations necessary to solve any query (*bucket-join* encompasses all *bucket* processing). Thus, the performance of the execution of a query tree may be improved by using multiple processors in the execution of each node. However, the storing of the intermediate results between nodes is normally considered to be the biggest time drain due to the time discrepancy between processor speed and the disk I/O time.

Pipelining is theoretically impossible for the binary operations. It is impossible

because the binary operations require all of the first relation to be compared with the second relation (or bucket of the relation). This means that one binary node has to be complete before the next node can finish. However, some of the initial processing in a binary node can be accomplished as pieces of the inputs are provided as results from a previous binary node. If the binary node uses the bucket type processing, the initial input blocks are hashed and blocks distributed to the processors where they will be processed. The processors take their initial hashed blocks and sort them to prepare for completely ordering the buckets of each of their inputs. The nested-loop processing does not sort or hash the tuples of the blocks it receives. But the nested-loop process can begin the comparison of blocks as long as the blocks are retained for comparison with all other blocks.

Pipelining is the type of parallelism that could most reduce the amount of secondary storage accesses in solving a query. However, pipelining does require processors to be committed to receiving results which removes a processor(s) from the current process, which may increase the time to perform the current process. The other consideration of parallelism is independent parallelism. The normal form of the query tree provides this opportunity because each subtree of a binary node can be performed independently. The ideal situation would have both subtrees of a binary node being processed concurrently and both finishing at exactly the same time, since the binary node cannot begin processing (see above for complete description) until both subtrees are complete. Therefore, independent parallelism is an important possibility in solving a query. Ideally, all forms of parallelism are used while processing the binary nodes of the query tree. However, the binary nodes require input and the initial nodes are required to first retrieve the base relations from secondary storage to begin the processing.

The initial nodes are data filters that retrieve the base relations from secondary storage. The processing of the initial nodes is not concerned about pipelining since this is the base level operation. This leaves only independent and intranode paral-

lelism. Intranode parallelism is easily and effectively implemented by using several processors. However, the retrieval of data from secondary storage has been the emphasis of research because the disk retrieval is time consuming. Intranode parallelism improves the performance of the retrieval only if the data has been distributed over the disks available. Distributing the relation to several disks allows the disks to simultaneously provide data to the processors performing the data filtering.

The initial nodes provide the opportunity for independent parallelism. However, for independent parallelism to be effective there must be available resources. If intranode parallelism is implemented to retrieve a portion of a relation from the secondary storage devices, then the storage devices are busy and are not available for independent parallelism. Therefore, if intranode parallelism is implemented, independent parallelism opportunities are reduced and vice versa. The next section examines some of the performance trade-offs of implementing parallelism.

### *8.5 Modeling a Multi-Step Query*

A multiple step query provides the opportunity for implementing parallelism. Therefore, the analytical modeling of individual operators is extended to a multi-step query. Modeling a query consisting of multiple steps requires some assumptions about resource allocation, interconnection of processor communication, and data placement in support of the query. Ideally, the query could apply parallelism to utilize the multiprocessor capability to decrease the execution time of the query. One method of executing the multi-step query is to execute each step separately and store the results for the next step. This method requires some method of storing the intermediate results, but allows all processors to concentrate on each step. One form of parallelism is pipelining. Pipelining allocates resources so the results of one step are passed directly to the next step. Previously, the theoretical consideration of the binary operations was discussed and it was said that the binary operations needed all complete inputs to accomplish the binary operation. This reduces the advantage of

pipelining but does not eliminate the use of pipelining in solving a multi-step query.

The modeling of a multi-step query combines effects of the control of resources, data distribution, individual operations, and the ability to communicate and pass information among the processors. Development of a multi-step model provides a basis for developing processor allocation, data distribution, and sequencing of events to provide the "best" performance. The first multi-step query to be modeled is an equi-join of two relations that have been reduced by prior selection and projection on the base relations. This query retrieves the two base relations, performs a self-proj on each relation and then performs a join. The environment assumes multiple processors, fully interconnected processor communication, multiple disks for storage of base relations, and temporary storage at processors when needed. This model is then extended to cover the more general query tree.

The utilization of parallelism to solve the defined multi-step query begins with parallel retrieval of the base relations. It is assumed that the base relations are distributed over several disks. It is also assumed that each processor used to retrieve the base relation has a corresponding disk. Thus, there is no contention for the disk resources. The model also assumes that the results are being sent to some outside data sink. The next consideration is how to allocate tasks to processors to accommodate pipelining and how much should tasks be separated. This leads to four different schemes for solving the multi-step query that joins two relations that have been reduced by selection and projection.

The schemes or models developed for the query use pipelining to provide overlapping processing of tasks where possible. Although previous models showed optimized retrieval performance when the relations are stored using indexed and ordered structures, the models developed here assume unordered-unindexed relations (the unordered-unindexed structure is the general retrieval case, other data structures may provide faster retrievals but the unordered-unindexed retrieval works for all cases). This means that the entire relation is retrieved and scanned to perform the

sel-proj operation. This method has shown that a linear time reduction is provided by providing more disk-processors for the selection operation. Therefore, it is known that the sel-proj operator is sensitive to the number of processors applied to the task.

The next step of the multi-step query is the joining of the two relations. The bucket join was shown to be the best algorithm to solve a equi-join. The first step of the bucket join is to distribute or hash the buckets to be joined. Therefore, the join is a two-step process, hashing and joining of buckets. This provides the allocation of two tasks to processors to perform the hashing and joining. This means that some processors could be used exclusively for hashing and some processors used for joining buckets. Another method of allocating tasks could have each processor hashing part of the input and when the hashing is complete and the buckets distributed, each processor would join its buckets. If the buckets for the join are too large for the processor memory, some of the tuples must be stored on secondary storage or the bucket can be hashed again to form smaller buckets with additional processors receiving the smaller buckets for processing. In the models presented here all of the tuples are considered to be stored on secondary storage available at each join processor.

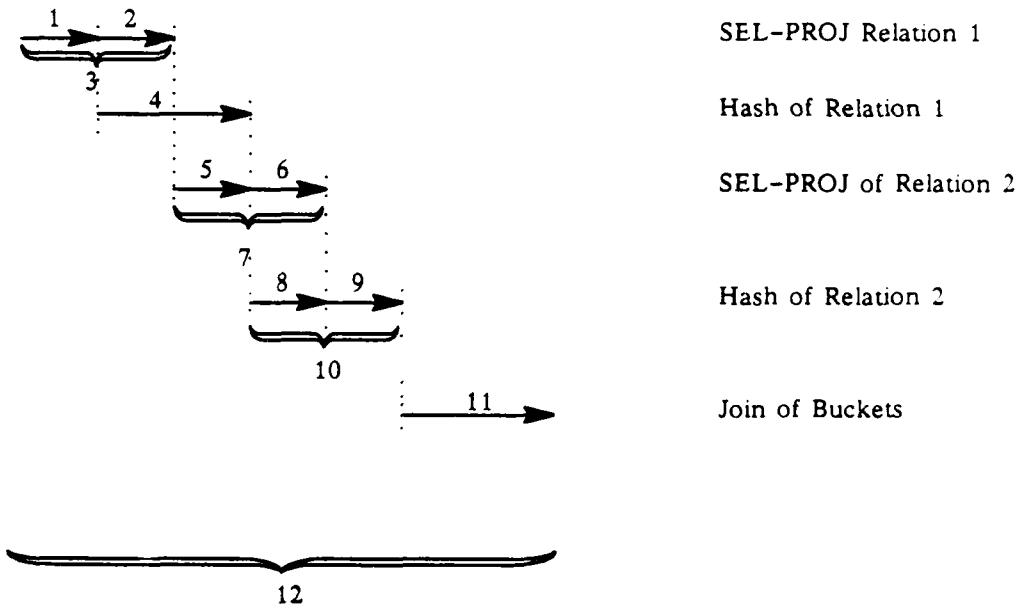
The situations described above provide four different cases to be examined for performing the example multi-step query. The four cases are: consecutive base relation retrieval with combined hash/join processors, concurrent base relation retrieval with hash/join processors, consecutive base relations with dedicated hash and join processors, and concurrent base relation retrieval and dedicated hash and join processors. The different cases apply the overlapping of processing in different manners. Figure 45 shows not overlapping the base relation retrievals and Figure 46 uses the overlapping retrieval of base relations. The overlap models show the assumption that each operation may be processed by several processors but at several points in the overlap model synchronization points. These points require all processors to

complete the current process before the next process can start. This means that the models of the individual steps presented in the following cases assume that the model is worst case time for a processor. These synchronization points also allow the assumed architecture to not impose some matching of hash processors to retrieval processors. An example of this is if there are 10 retrieval processors but only 2 hash processors, the retrieval processors may have to wait for the hash processors. The model accounts for this by using the time of the longest process up to the synchronization point. Therefore, if the hash takes longer than the retrieval/sel-proj. then the time of the hash is the dominating factor that is used to compute the total execution time. The parameters to be varied in the cases are the number of processors allocated to each task, the separation of the join tasks, and the overlapping of relation retrievals.

#### *8.5.1 Multi-Step Query Processing using Consecutive Relation Retrieval.*

The base relation retrieval in Case 1 first retrieves the entire first relation, performs the sel-proj operation and hashes the relation to the join processors. The second relation is then retrieved, operated on, and hashed. The join can then be performed. Figure 45 shows the potential overlapping of actions when the relations are retrieved consecutively. This configuration assumes that the base relations are distributed across all the disks (assuming the blocks of the relation are greater than the number of disks). Therefore, all of the retrieval processors are used to retrieve the relations.

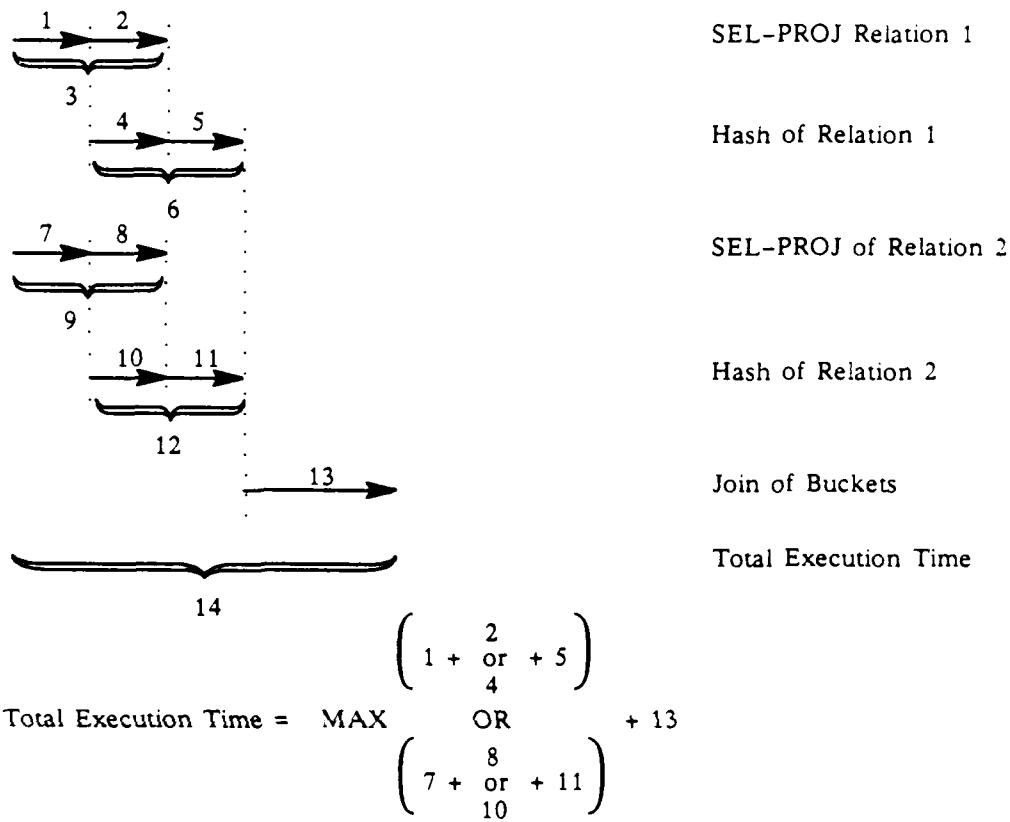
The processing in Case 1 continues by using all of the processors allocated for the hash/join processing to hash the inputs. This implies that there are separate processors for retrieving and joining the relations. The processors assigned to join the relations implement the join by grouping the relations into disjoint sets of tuples and then performing the join on the disjoint set of tuples. With multiprocessors, each processor can then either hash the input into buckets, hash buckets or both hash and join. This case assigns the join processors to both hash part of the inputs and then join a group of tuples. If the buckets for the join are too large for the processor



$$\text{Total Execution Time} = 1 + \frac{4}{(2+5)} \text{ or } + \frac{6}{8} \text{ or } + 9 + 11$$

- 1 - Time to first block of output --  $T_{isp}$
- 2 - Time to complete SEL-PROJ after first results produced --  $T_{sp} - T_{isp}$
- 3 - Time for complete SEL-PROJ =  $1 + 2$  --  $T_{sp}$
- 4 - Time to hash result of SEL-PROJ of Relation 1 --  $T_h$
- 5 - Time to first block of results --  $T_{isp}$
- 6 - Time to complete SEL-PROJ after first results produced --  $T_{sp} - T_{isp}$
- 7 - Time for SEL-PROJ of Relation 2 =  $5 + 6$  --  $T_{sp}$
- 8 - Time of Hash to point of receiving last block to be hashed --  $T_{h2}$
- 9 - Time to complete Hash after receiving last block to be hashed --  $T_h - T_{h2}$
- 10 - Time to hash results of SEL-PROJ of Relation 2 --  $T_h$
- 11 - Time to join individual buckets to complete Join --  $T_j$
- 12 - Total execution time

Figure 45. Process Overlapping of Multi-Step Query



- 1 - Time to first block of results --  $T_{1sp}$
- 2 - Time to complete SEL-PROJ after first results produced --  $T_{sp} - T_{1sp}$
- 3 - Time for complete SEL-PROJ =  $1 + 2$  --  $T_{sp}$
- 4 - Time of Hash to point of receiving last block to be hashed --  $T_{hc2}$
- 5 - Time to complete Hash after receiving last block to be hashed --  $T_{hc} - T_{hc2}$
- 6 - Time to hash result of SEL-PROJ of Relation 1 --  $T_{hc}$
- 7 - Time to first block of results --  $T_{1sp}$
- 8 - Time to complete SEL-PROJ after first results produced --  $T_{sp} - T_{1sp}$
- 9 - Time for SEL-PROJ of Relation 2 =  $7 + 8$  --  $T_{sp}$
- 10 - Time of Hash to point of receiving last block to be hashed --  $T_{hc2}$
- 11 - Time to complete Hash after receiving last block to be hashed --  $T_{hc} - T_{hc2}$
- 12 - Time to hash results of SEL-PROJ of Relation 2 --  $T_{hc}$
- 13 - Time to join individual buckets to complete Join --  $T_j$
- 14 - Total execution time

Figure 46. Process Overlapping of Multi-Step Query with Overlapping Relation Retrievals

memory, some of the tuples must be stored on secondary storage or the bucket can be hashed again to form smaller buckets with additional processors receiving the smaller buckets for processing. In the models presented here all of the tuples are considered to be stored on secondary storage available at each join processor.

The multi-step query model is based upon the individual operator models developed in previous chapters. But the overlapping of process requires some intermediate time parameters of operators to be computed. The first intermediate time parameter needed is the time of the retrieval/sel-proj operation to produce the first block of results. The time to produce the first block of results depends upon the number of blocks that must be scanned to produce a block of results. Therefore, it must be determined how many blocks of the base relation must be scanned by the sel-proj to produce one block of results. The formula for determining the total number of blocks of results that are produced by the retrieval/sel-proj operation (using the parameters of Table 2) is:

$$(v * (R * f) * (B/r) * (1/B)) \quad (207)$$

The value needed, however, is the number of blocks,  $x$ , to produce one block of results. Using the formula of above, the equation for finding the number of blocks to be scanned is:

$$\frac{(v * (R * f) * (B/r) * (1/B))}{R} = \frac{1}{x} \quad (208)$$

This reduces to

$$x = \frac{(v * (R * f) * (B/r) * (1/B))}{R} \quad (209)$$

which further reduces to

$$x = r/(v * f) \quad (210)$$

Thus, the time required by the retrieval/sel-proj operation at each retrieval processor to produce and send the first block of results to the hash processors is:

( $T_{1,sp}$  is 1 and 5 in Figure 45 and 1 and 7 in Figure 46)

$$T_{1,sp} = T_m + T_d + T_{io} + \max \left| \begin{array}{l} ((r/(v*f)) * T_{sc}) \\ \text{or} \\ (((r/(v*f))/b) - 1) * T_s \\ +(r/(v*f)) * T_{io} \end{array} \right| + T_{bt} \quad (211)$$

The total time for the sel-proj/retrieval operation (using p processors with p disks) is:

( $T_{sp}$  is 3 and 7 in Figure 45 and 3 and 9 in Figure 46)

$$T_{sp} = T_m + T_d + T_{io} + \max \left| \begin{array}{l} (((R/p) - 1) * T_{sc}) \\ +(((v * ((R/p) * f) * (B/r) * (1/B)) - 1) * T_{bt} \\ \text{or} \\ (((R/p)/b) - 1) * T_s \\ +(R/p) * T_{io} \end{array} \right| + T_{sc} + T_{bt} \quad (212)$$

The blocks of the results produced,  $((v * (R * f) * (B / r) * (1 / B))$ , will be referred to as  $R_1$  and the results produced from the sel-proj of the second relation, S, will be called  $S_1$  equals  $((v * (S * f) * (B / s) * (1 / B))$ . The inputs into the hash processors is then equal to the number of blocks divided by the number of processors performing the hash ( $p_h$  which is 4 and 10 in Figure 45).

The hash process consists of receiving the input from the sel-proj operation, scanning the blocks to hash the tuples, and distributing the hashed buckets. The hash processors for Case 1 must also receive buckets from other hash processors for later join processing. Therefore, the hash processors hashes, sends blocks of information, and receives blocks of hashed information. The hashing function is assumed to produce approximately evenly distributed buckets. The functions to determine the number of blocks to be sent to each other processor for this case is:

$$R_2 = (R_1 / (p_h - 1)) + 1 \text{ for the first relation and} \quad (213)$$

$$S_2 = (S_1 / (p_h - 1)) + 1 \text{ for the second relation} \quad (214)$$

The time to process the hash, send the buckets, and receive the buckets to be processed for the input  $R_1$  is:

$$T_h = T_m + (R_1 * (T_{bt} + T_{sc})) + ((2 * (p_h - 1) * R_2) * T_{bt}) \quad (215)$$

Another consideration is that the blocks retained or received by the processor might exceed the memory capacity of the processor. Then some of the blocks must be sent to secondary storage. During the hashing of the first relation, if  $R_2 > p_b$ , the additional segment added to the performance model is:

$$+ T_d + ((R_1 - (p_b - 2)) * T_{io}$$

During the hashing of the second relation the additional segment is:

$$\begin{aligned} \text{if } & (R_2 > p_b) \\ & + T_d + (S_2 * T_{io}) \\ \text{else } & \\ & + T_d + ((R_2 - (p_b - S_2 - 2)) * T_{io}) \end{aligned}$$

Also, the time used in the hash until the final block is received must be modeled. The actions required after the last block is received are hash the block and distribute the final block of each bucket to each join processor. Using this, the time to the point of receiving the last block to be hashed is:

( $T_{h2}$  is 9 in Figure 45.)

$$T_{h2} = T_m + ((R_1 - 1) * (T_{bt} + T_{sc})) + ((2 * (p_h - 1) * (R_2 - 1) * T_{bt}) \quad (216)$$

The final operation is the joining of the distributed buckets. The performance model where each processor joins input relations of size  $R_2$  and  $S_2$  and tuple sizes of  $r_{size}$  and  $s_{size}$  is:

( $T_j$  - the time to perform the join is 11 in Figure 45 and 13 in Figure 46)

if  $((R_2 + S_2) < p_b)$

$$T_j = T_m + T_{sort}(Tb, R_2, B/r_{size}) + T_{sort}(Tb, S_2, B/s_{size}) + (((R_2 + S_2)/2) * T_b) + (jB * T_{bt}) \quad (217)$$

else if  $(R_2 < p_b)$

$$T_j = T_m + T_{sort}(Tb, R_2, B/r_{size}) + T_d + (R_2 * T_{io}) + T_d + ((S_2 - (p_b - 2)) * T_{io}) + T_{sort}(Tb, S_2, B/s_{size}) + (((R_2 + S_2)/2) * T_b) + T_d + (R_2 * T_{io}) + (jB * T_{bt}) \quad (218)$$

else

$$+ T_d + ((p_b - (S_2 - (p_b - R_2 - 2))) * T_{io}) + T_{sort}(Tb, (p_b - 2), t_b)$$

$$+ ((S_2 - (p_b - 2))/p_b) * (T_d + (p_b * T_{io}) + T_{sort}(Tb, p_b, t_b) + T_i + (p_b * T_i))$$

$$+ T_d + (S_2 - ((S_2 - (p_b - 2))/p_b) * T_{io})$$

$$+ T_{sort}(Tb, (S_2 - (S_2 - (p_b - 2))/p_b), t_b)$$

$$+ \max \left| \begin{array}{l} ((S_2 - (p_b - (S_2 - (S_2 - (p_b - 2))/p_b))) * (T_i + T_{io})) \\ \quad + (S_2 * (T_d + T_{io})) \\ \quad \text{or} \\ \sum_{i=2}^{((S_2 - (p_b - 2))/p_b) + 1} (((p_b/2) * i) * T_b) \end{array} \right|$$

$$+ 2 * T_d + 2 * T_{io} + \max \left| \begin{array}{l} (((R_2 + S_2)/2) - 1) * T_s \\ \quad + ((jB - 1) * T_{bt}) \\ \quad \text{or} \\ (((R_2 + S_2) - 2) * (T_i + T_{io})) \end{array} \right| \quad (219)$$

$$+ T_b + T_d + T_{io} + T_{st}$$

else  $R_2 > p_b$

$$T_j = T_m + T_{sort}(Tb, (p_b - 2), (B/r_{size})) + T_i + ((p_b - 2) * T_i)$$

$$\begin{aligned}
& + ((R_2 - (p_b - 2))/p_b * (T_d + (p_b * T_{io}) + T_{sort}(T_b, p_b, (B/r_{size})) + T_i \\
& + (p_b * T_{io}))) + T_d + (R_2 - (R_2 - (p_b - 2))/p_b * T_{io}) \\
& + T_{sort}(T_b, (R_2 - (R_2 - (p_b - 2))/p_b), (B/r_{size})) \\
& \quad \left| \begin{array}{l} (R_2 * (T_d + T_{io})) + (R_2 * (T_d + T_{io})) \\ \text{or} \\ \sum_{i=2}^{((R_2 - (p_b - 2))/p_b) + 1} (((p_b/2) * i) * T_b) \end{array} \right| \\
& + \max \left| \begin{array}{l} \text{if } (S_2 < (p_b - 2)) \\ + T_d + (S_2 * T_{io}) + T_{sort}(T_b, S_2, B/s_{size}) \\ + (((R_2 + S_2)/2) * T_b) + T_d + (R_2 * T_{io}) + (jB * T_{bt}) \end{array} \right| \quad (220) \\
& \text{else} \\
& + (S_2/p_b) * (T_d + (p_b * T_{io}) + T_{sort}(T_b, p_b, B/s_{size}) + T_d \\
& + (p_b * T_{io})) + T_d + ((S_2 - (S_2/p_b) * T_{io}) + T_{sort}(T_b, (S_2 - (S_2/p_b)), B/s_{size}) \\
& \quad \left| \begin{array}{l} (S_2 * (T_d + T_{io})) + (S_2 * (T_d + T_{io})) \\ \text{or} \\ \sum_{i=2}^{(S_2/p_b)} (((p_b/2) * i) * T_b) \end{array} \right| \\
& + \max \left| \begin{array}{l} (((((R_2 + S_2)/2) - 1) * T_b) \\ + ((jB - 1) * T_{bt}) \\ \text{or} \\ (((R_2 + S_2) - 2) * (T_d + T_{io})) \end{array} \right| \\
& + 2 * T_d + 2 * T_{io} + \max \left| \begin{array}{l} T_b + T_d + T_{io} + T_{bt} \end{array} \right| \quad (221)
\end{aligned}$$

where

$$jB = jsf * R_1 * (B/r_{size}) * S_2 * [r_{size} + s_{size}] / s_{size} \quad (222)$$

and  $T_{sort}(T_b, p_b, t_b)$  is:

$$(T_b * (t_b * p_b) * \log_2(t_b * p_b)) / (t_b * \log_2 t_b) \quad (223)$$

The time to complete the multi-step query can now be modeled by combining the individual components providing for the overlapping actions as shown in Figure 45. The performance model for Case 1 when the relations are retrieved individually and the hash processors also perform the join is:

M - 1

$$\begin{aligned}
 \text{Case1} = & T_{1sp}(R) + \max \left| \begin{array}{c} (T_{sp}(R) - T_{1sp}(R)) + T_{1sp}(S) \\ \text{or} \\ T_h(R_1) \end{array} \right| \quad (224) \\
 & + \max \left| \begin{array}{c} (T_{sp}(S) - T_{1sp}(S)) \\ \text{or} \\ T_{h2}(S_1) \end{array} \right| + (T_h(S_1) - T_{h2}(S_1)) + T_j(R_2, S_2)
 \end{aligned}$$

**8.5.2 Multi-Step Query Processing using Concurrent Relation Retrieval.** The second case assumes a different data distribution. This case assumes that the relations are distributed so they can be retrieved concurrently. This illustrates a different data placement of alternating disks to allow concurrent retrievals of different relations. This concurrent retrieval also forces the hashing processors to be split so that one half of the processors hash the first relation and the others hash the second relation. The relations are hashed to all of the processors for the join. Figure 46 illustrates the overlapping of actions for this case.

The performance models for this case require the hashing process model to be altered slightly to reflect the hashing the first relation while receiving buckets of both relations for the later join. Therefore, the hashing process model for input  $R_1$  (assuming concurrent hashing of  $S_1$  and  $p_{h2}$  processors performing the hash of the other relation) is:

( $T_h$  is 6 and 12 in Figure 46)

$$T_{h1} = T_m + (R_1 * (T_{k1} + T_{k2})) \quad (225)$$

$$\begin{aligned}
& + T_{sc})) + (((p_h + p_{hx} - 1) * ((R_1 / ((p_h + p_{hx}) - 1) + 1) * T_{bt}) \\
& + ((p_h - 1) * ((R_1 / ((p_h + p_{hx}) - 1) + 1) * T_{bt}) \\
& + (p_{hx} * ((R_2 / ((p_h + p_{hx}) - 1) + 1) * T_{bt})
\end{aligned}$$

and ( $T_{hc2}$  is 4 and 10 in Figure 46)

$$\begin{aligned}
T_{hc2} = & \quad T_m + ((R_1 - 1) * (T_{bt} \\
& + T_{sc})) + (((p_h + p_{hx} - 1) * (R_1 / ((p_h + p_{hx}) - 1) * T_{bt}) \\
& + ((p_h - 1) * (((R_1 / ((p_h + p_{hx}) - 1) * T_{bt}) \\
& + (p_{hx} * ((R_2 / ((p_h + p_{hx}) - 1) * T_{bt})
\end{aligned} \tag{226}$$

The additional segment to account for the time to store information on secondary storage for the processors hashing relation R is:

$$\begin{aligned}
& \text{if } (R_2 > p_b) \\
& \quad + ((R_2 - p_b - 2) * (T_{io} + T_d)) + (S_2 * (T_d + T_{io})) \\
& \text{else} \\
& \quad + T_d + ((R_2 - (p_b - S_2 - 2)) * T_{io})
\end{aligned}$$

and for the processors hashing the the relation, S, it is:

$$\begin{aligned}
& \text{if } (S_2 > p_b) \\
& \quad + ((S_2 - p_b - 2) * (T_{io} + T_d)) + (R_2 * (T_d + T_{io})) \\
& \text{else} \\
& \quad + T_d + ((S_2 - (p_b - R_2 - 2)) * T_{io})
\end{aligned}$$

The equation expressing the performance of the multi-step query when the relations are on alternate disks providing concurrent relation retrieval (remembering half of the retrieval processors and half of the hash processors are dedicated to each relation) is:

M - 2

$$\begin{aligned}
 Case 2 = \max & \left| \begin{array}{c} T_{1sp}(R) + \max \left| \begin{array}{c} (T_{sp}(R) - T_{1sp}(R)) \\ or \\ T_{hc2}(R_1) \end{array} \right| + (T_{hc}(R_1) - T_{hc2}(R_1)) \\ or \\ + T_{1sp}(S) + \max \left| \begin{array}{c} (T_{sp}(R) - T_{1sp}(R)) \\ or \\ T_{hc2}(S_1) \end{array} \right| + (T_{hc}(S_1) - T_{hc2}(S_1)) \end{array} \right| + T_j(R_2, S_2)
 \end{aligned} \tag{227}$$

*8.5.3 Multi-Step Query Processing using Consecutive Relation Retrieval with Dedicated Join Processors.* The third method of assigning tasks to processors to complete the multi-step query is to divide the tasks of hashing and processing. This scheme dedicates processors to only hashing and other processors to only performing the join. The significance of this method is that the hashing processors do not have to be interrupted to receive buckets for later joining. Figure 47 shows a logical view of this processing environment and Figure 45 shows the overlap of processing possible. This case repeats the view of data distribution of Case 1. First, all the retrieval processors retrieve and perform the sel-proj on the first relation. Then the second relation is retrieved and processed. This is the same processing as Case 1 and uses the same model of the retrieval/sel-proj operation.

The hash processors first concentrate on the first relation, hashing buckets to the join processors. The hash processors only hash and send buckets to the join processors since there are separate processors assigned for each portion of the join processing - hashing inputs into buckets and joining buckets. Therefore, the performance model of hashing that does not have to account for receiving buckets, where there are  $p_h$  join processors, is:

$$T_{ho} = T_m + (R_1 * (T_{bt} + T_{sc})) + (((p_h - 1) * R_2) * T_{bt}) \tag{228}$$

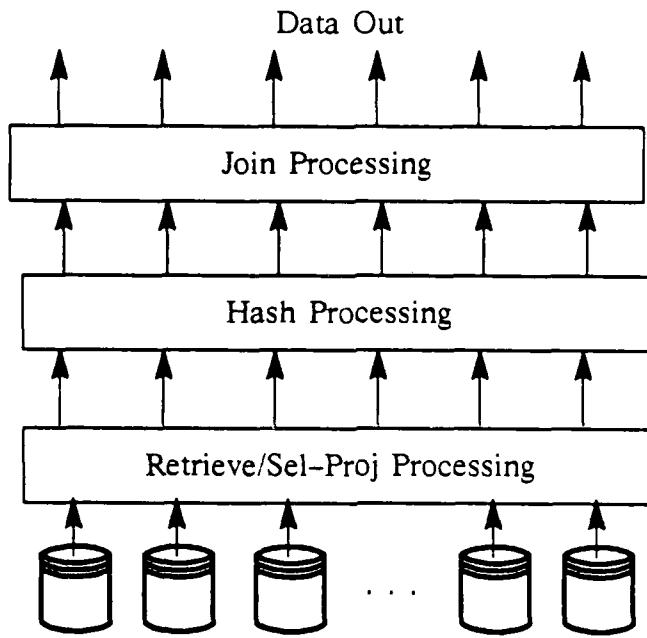


Figure 47. Logical View of Case 3 Multi-Step Query Processing

where the output sizes going to each join processor are:

$$R_2 = (R_1/(p_j - 1)) + 1 \quad (229)$$

$$S_2 = (S_1/(p_j - 1)) + 1 \quad (230)$$

and the time of the hash to the point of receiving the last block is:

$$T_{ho2} = T_m + ((R_1 - 1) * (T_{bt} + T_{sc})) + (((p_h - 1) * (R_2) - 1) * T_{bt}) \quad (231)$$

The join processors receive the buckets from the hash processors and if necessary store data on secondary storage. One concept the following performance model does not reflect is initial processing of buckets received while still receiving other buckets. This situation could allow the join processor to do the sorting of the buckets of the first relation while the second relation is being hashed and processed. This initial processing to prepare the relation for the join is not included in the models because any estimates of the time required do not adequately model the interaction of receiving the blocks and performing the initial processing. Also, for comparison

purposes it is considered to be more sound to evaluate the worst case of this situation. Therefore, the performance model of the join is the same as the join processing model described previously,  $T_j$ , with the following addition to the beginning of the join model.

$$T_{jo} = ((R_2 + S_2) * T_{bt}) + T_j \quad (232)$$

The model is also extended with the following segment when  $R_2 + S_2 > p_b$

$$+ ((R_2 + S_2 - p_b) * (T_d + T_{io}))$$

The complete model for consecutive relation retrievals using dedicated join processors (using the overlap model of Figure 45) is:

### M - 3

$$\begin{aligned} Case3 &= T_{1sp}(R) + \max \left| \begin{array}{c} (T_{sp}(R) - T_{1sp}(R)) + T_{1sp}(S) \\ \text{or} \\ T_{ho}(R_1) \end{array} \right| \\ &\quad + \max \left| \begin{array}{c} (T_{sp}(S) - T_{1sp}(S)) \\ \text{or} \\ T_{ho2}(S_1) \end{array} \right| + (T_{ho}(S_1) - T_{ho2}(S_1)) + T_j(R_2, S_2) \end{aligned} \quad (233)$$

*8.5.4 Multi-Step Query Processing using Concurrent Relation Retrieval with Dedicated Join Processors.* The final case uses the same processing and data distribution scheme as Case 2 with dedicated hash and join processors. Figure 46 illustrates the overlapping of processes for this case. Assuming that each relation is processed by half of the retrieval and hash processors, the following performance model is developed:

M - 4

$$Case 4 = \max \left\{ \begin{array}{l} T_{1sp}(R) + \max \left| \begin{array}{c} (T_{sp}(R) - T_{1sp}(R)) \\ or \\ T_{ho2}(R_1) \end{array} \right| + (T_{ho}(R_1) - T_{ho2}(R_1)) \\ or \\ + T_{1sp}(S) + \max \left| \begin{array}{c} (T_{sp}(R) - T_{1sp}(R)) \\ or \\ T_{ho2}(S_1) \end{array} \right| + (T_{ho}(S_1) - T_{ho2}(S_1)) \end{array} \right\} + T_{jo}(R_2, S_2) \quad (234)$$

**8.5.5 Multi-Step Modeling Results.** The multi-step query provides the opportunity for intranode parallelism and pipelining. The four cases developed use the parallelism concepts in different manners. Two main points are evaluated by the multi-step models: relation storage distribution and process allocation.

The first point – the distribution of relations on secondary storage for retrieval – addresses whether concurrent retrieval of relations is better than separate retrievals of the relations. If a relation is distributed on all disks, all the retrieval processors can retrieve the relation. If only half the disks and processors are used to retrieve the relation, each processor must do twice the work. It seems that either method should be approximately equal in performance; however, when one relation is much larger than the other (such as 10 times as large), the workload is not evenly distributed because one half of the processors retrieve a small number blocks (such as 1/10 the blocks). This means that one set of processors finishes before the other group of processors.

The second evaluation point is dedicating processors to tasks. The models portray processors working on two tasks at different points of the processing and dedicating the tasks to a processor for the entire processing cycle. The implication of dedicated tasks is that the processor may perform some initial processing of the data while still needing data before the process can begin (See Case 3 above). This

illustrates the theoretical consideration that the binary operations cannot begin the actual processing of the operation until the entire two inputs are present. But this does not stop initial processing of one relation to place the relation in a preferred structure while waiting for the entire input to become available.

The performance of the four models using the performance parameters used in the performance modeling of the individual operations is reflected in Figures 48 through 79. The performance values presented were computed using the same performance parameters as the individual operator's performance models. The results assume that processors are connected by a fully interconnected network to provide processor communication.

The focus of the multi-step query is to determine the best method of task allocation. Therefore, all of the performance models were computed with a processor allocation that assumes the number of join and hash processors totals 20 and the number of retrieval processors can be adjusted without affecting the number of hash and join processors. This evaluates the processor allocation strategy. Figures 48 and 49 (also see Appendix A) show the comparison between consecutive and concurrent relation retrieval. The result of these models is that the consecutive relation processing performs better than the concurrent relation retrieval. The following section discusses concurrent versus consecutive retrieval.

Figures 50 and 51 and the graphs presented in Appendix A compare the different cases that use consecutive retrievals. The notation in the charts is "Case 1 x-y" or "Case 3 x-y-z". This means that for Model M-1 or Case 1, x is the number of processor/disk pairs used to store and retrieve the relations and y processors hash/join the results of the sel-proj. Model M-3 or Case 3 x-y-z means consecutive retrieval of the relations for processing, retrieving the relation with x processors, hashing the relation with y processors, and joining the relations with z processors.

The results presented do not provide a clear cut best method of task allocation. However, the results do illustrate the important element of control - balance.

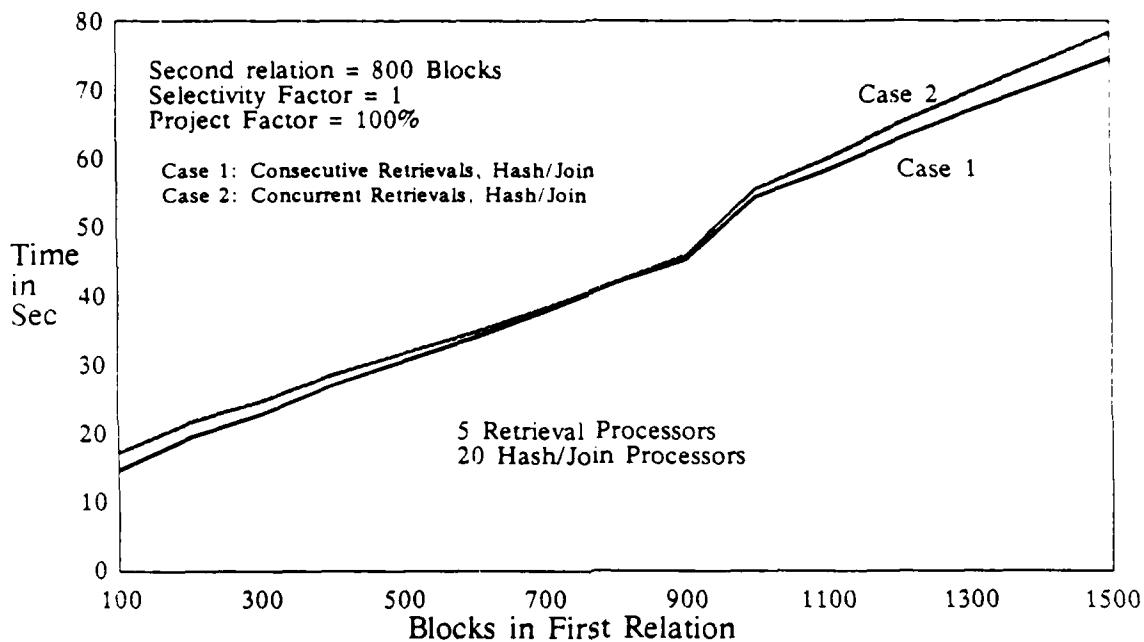


Figure 48. Multi-Step Performance using Hash/Join

The results showed that for the large input sizes the hash/join processing performs better because it balances the workload over all available processors. Therefore for each retrieval processor size two cases are presented, showing a smaller size and larger size for the fixed input relation size. When the input sizes into the hash/join processing are decreased by the selectivity and projection factors, the separation of tasks provides slightly better results because the hashing and join are balanced to the resources available.

#### *8.6 Extension of Model to Multi-Binary Node Queries*

The previous multi-step query models only considered a single join operation. This model needs to be extended to consider modeling queries with multiple binary operations. This model extension is based on the normal form query tree. The query tree provides the key to developing this model – a recursive definition. The query tree is a binary tree. Therefore, the performance model starts at the root of the tree. The root of the tree is a binary node (assuming no duplicate removal) that joins two

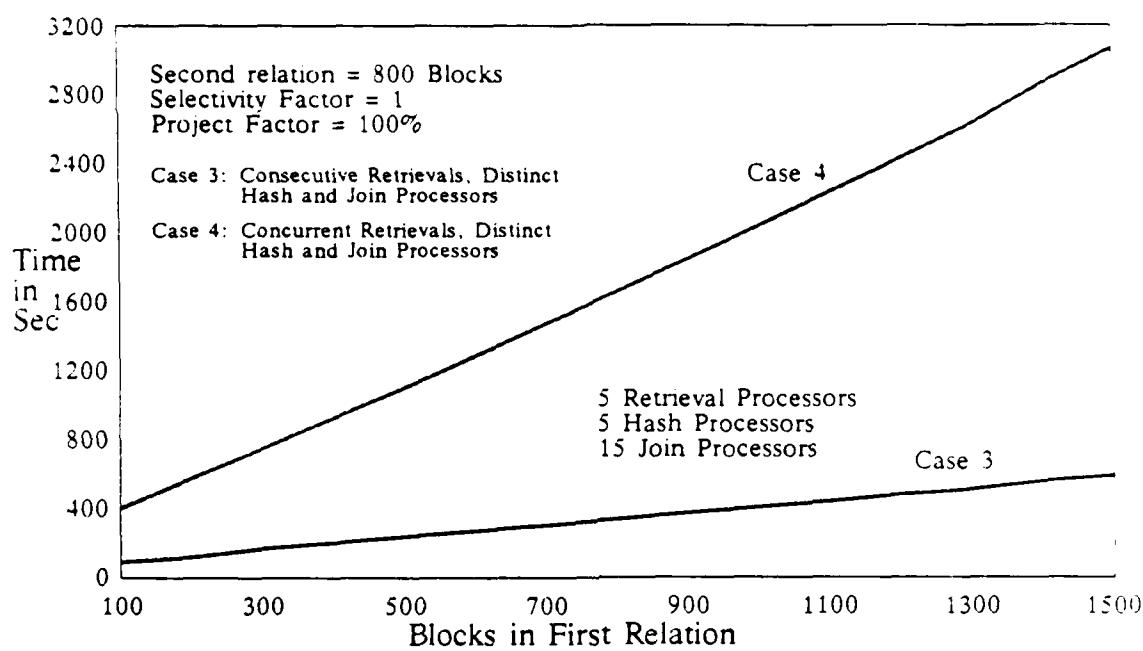


Figure 49. Multi-Step Performance using Separate Hash and Join

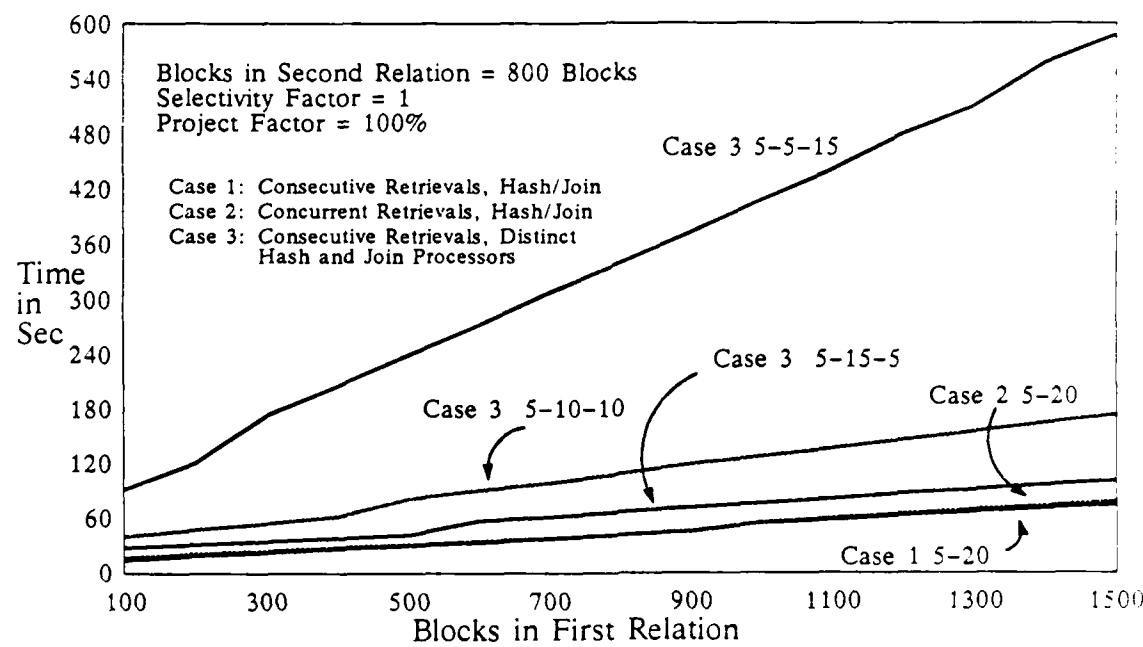


Figure 50. Performance with Selectivity Factor = 1, Projectivity Factor = 100% using 5 Retrieval Processors - 1

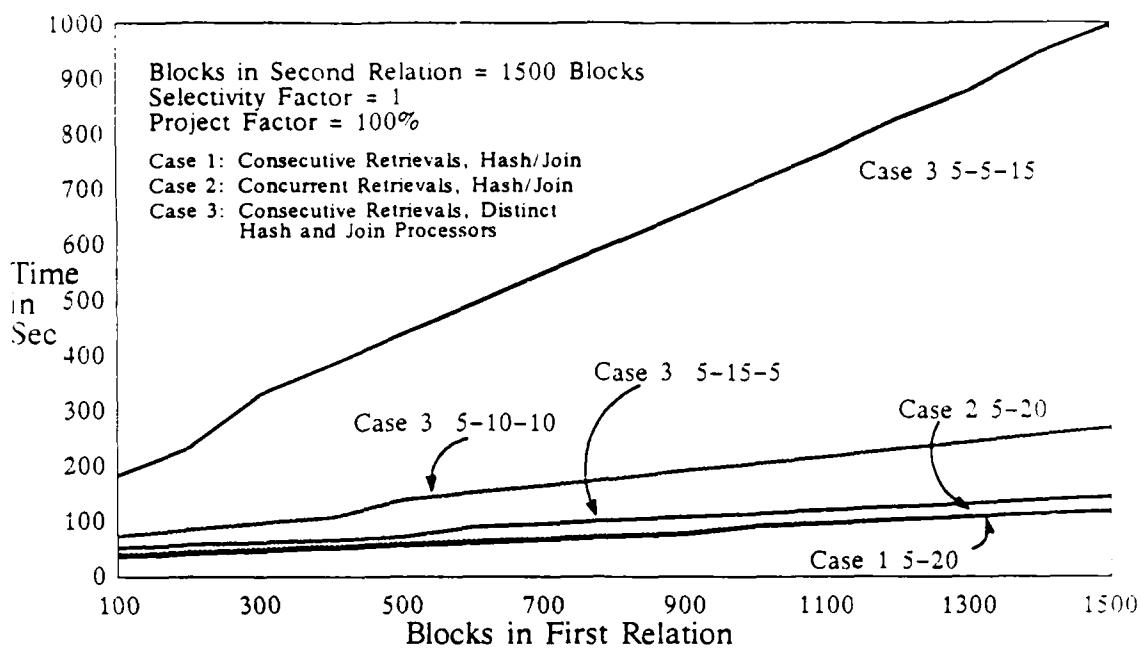
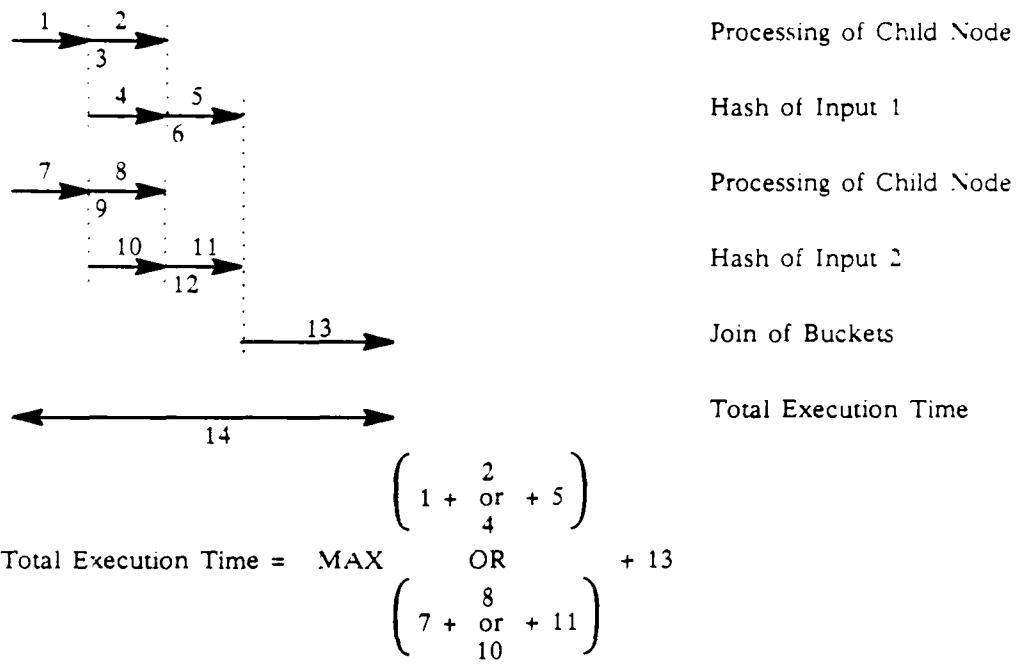


Figure 51. Performance with Selectivity Factor = 1, Projectivity Factor = 100% using 5 Retrieval Processors - 2

relations that are produced the child nodes of the root. Figure 52 shows the overlap of processing at the root node. A child node may be the root of a subtree that also joins two relations produced by child nodes. The child node then is the root node of a tree. Therefore, the performance model recursively finds the time to execute each subtree. Figure 52 shows how the processing of the root node depends upon the processing of the child nodes.

The lowest level subtree of the multi-join query is the multi-step query modeled in the previous section. Therefore, if a multi-join query consisted of joining four base relations, the query tree would have two subtrees to the root join and the subtrees would be the multi-step query model of the previous section that retrieves the inputs from secondary storage. The root join receives the inputs from the subtrees, hashes the inputs to form the buckets, and then joins the buckets. The hash processing begins as soon as the first block of input is received and is overlapped with the processing of the subtrees.



- 1 - Time to first block of results
- 2 - Time to complete child node after first results produced
- 3 - Time for complete child node execution = 1 + 2
- 4 - Time of Hash to point of receiving last block to be hashed
- 5 - Time to complete Hash after receiving last block to be hashed
- 6 - Time to hash result of child node 1
- 7 - Time to first block of results
- 8 - Time to complete child node after first results produced
- 9 - Time for complete child execution = 7 + 8
- 10 - Time of Hash to point of receiving last block to be hashed
- 11 - Time to complete Hash after receiving last block to be hashed
- 12 - Time to hash results of child node 2
- 13 - Time to join individual buckets to complete Join
- 14 - Total execution time

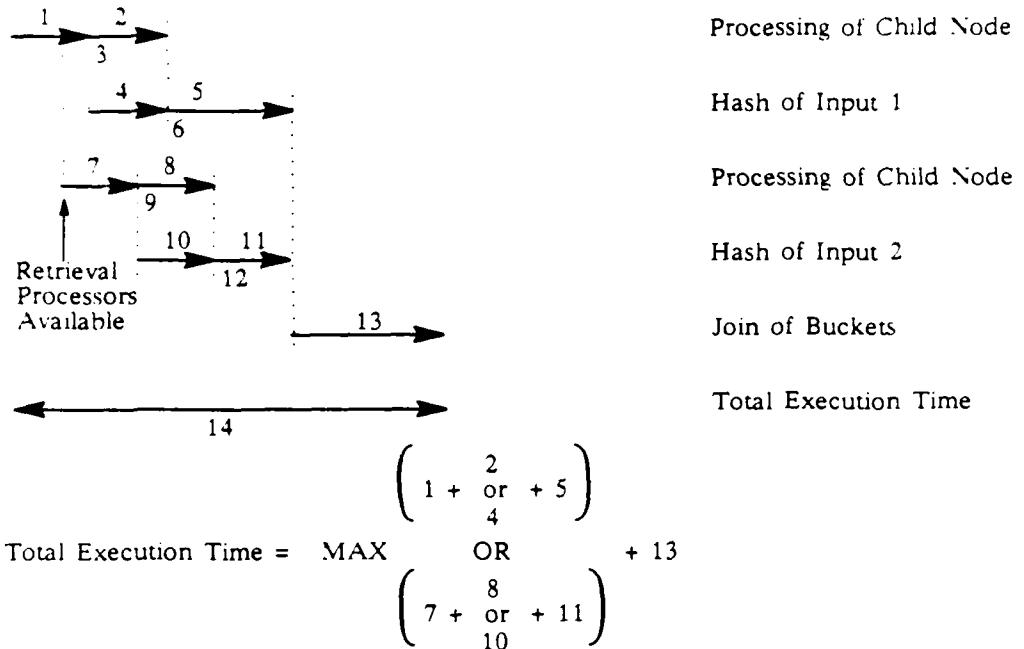
Figure 52. Processing Overlap Model

The previous multi-step query model considered two possible data distributions - a relation stored on all disks or relations stored on half of the disks. When the multi-step query model is extended to include multi-joins, the data distribution could force the start of a subtree processing to be delayed until the retrieval processors and disks become available. This may constrain the join processing because of the delay of receiving the input from one of the subtrees. Figure 53 shows the effects of this delayed processing.

The multi-join model also may be affected by the task allocation of the processors. The performance results of the previous sections need to be extended to account for the multi-join model where the root node depends upon the processing of the subtrees. The multi-step query of the previous section is used as the subtree models that provide the inputs for another binary node that joins the two inputs. The root node that provides the last join does not have the ability to predict the sequencing of events like the nodes that join the base level relations. Therefore, the join here assumes that hash processors are divided to hash the two inputs simultaneously. The model parameters to be varied are the data distribution ( a relation is assumed to be stored on each disk or on 1/4 of the disks for the results presented) and the allocation of tasks in the binary node.

The binary node is a combination of operators that join the inputs and then perform a sel-proj on the join results. The allocation of tasks assumes that the processors performing the join of the buckets also perform the sel-proj operations. Thus, the allocation of tasks refers to the use of using the same processors for both hashing and joining buckets or using separate processors for hashing the inputs and other processors to join the buckets.

The results (see Appendix B) show that the performance may vary greatly with the size of the inputs of each node. This emphasizes the concept of balancing the process to the inputs provides the best performance. The performance of some of the instances are better when the base relations are not distributed on all the disks.



- 1 - Time to first block of results
- 2 - Time to complete child node after first results produced
- 3 - Time for complete child node execution = 1 + 2
- 4 - Time of Hash to point of receiving last block to be hashed
- 5 - Time to complete Hash after receiving last block to be hashed
- 6 - Time to hash result of child node 1
- 7 - Time to first block of results
- 8 - Time to complete child node after first results produced
- 9 - Time for complete child execution = 7 + 8
- 10 - Time of Hash to point of receiving last block to be hashed
- 11 - Time to complete Hash after receiving last block to be hashed
- 12 - Time to hash results of child node 2
- 13 - Time to join individual buckets to complete Join
- 14 - Total execution time

Figure 53. Processing Overlap Model with Retrieval Processor Constraint

However, Case 1 and Case 3, which use complete distribution of the base relations, provides either the "best" performance or very close to the "best" performance for the given set of performance parameters used.

### *8.7 Control of Resources and Task Allocation*

The multi-step query models presented use data-flow to control the processing of the query. Data-flow assumes that when the data becomes available, a process can start. This implies a fixed allocation of processors to tasks in the performance models. But this may leave some processors idle for lengthy periods of time. Therefore, the following section discusses how the performance models developed can be used to support control strategies.

The assignment of resources and the allocation of tasks to the resources constitutes the control structure of the system. The results presented for the multi-step query illustrate the problems the controlling the environment to provide the best possible performance. First, the main elements of the control structure are defined. Then these elements are used to define the considerations of control and how they might be applied to the multi-step query processing.

The main elements of the control structure for executing a query are:

- Resources available
  - number of processors with some memory capability
  - interprocessor communication capability
  - secondary data storage capability
- Steps required to solve query
- Size of relations used in processing

The goal of the control structure is to perform each query as quickly as possible. This means when multiple processors are available it would be desirable

incorporate parallelism to improve performance. The first consideration in using the resources available is the size of the problem or, put another way, the size of the input relation(s) into a relational operator in a database query. The goal of the processing is to use all processors and to have each processor share equally in the workload by doing  $1/p$  of the total process (assuming  $p$  processors). However, using this concept requires prior knowledge of the size of the input relations in order to match the resources available to the processing required. For a single step query, this prior knowledge is available. In a multi-step query, the only knowledge of input sizes is for the first retrieval processes. The input sizes for later steps can only be related to the worst or expected case size of input. This stresses the control ability to allocate adequate resources without wasting resources that could be used for a different task. (Dynamic control based on real-time reporting of intermediate relation sizes is not considered here, but may be worth investigating.)

The multi-step model results presented above illustrate the problem of balancing resources to the problem size. The worst case scenario presented in one step requires the entire relations to be passed through the sel-proj operation. This situation provides the best performance when the available resources are allocated to each phase of the hash/join operation.

However, at some point adding more processors will not improve the performance because the communication overhead will exceed the performance improvement. This is the case in the ideal situation is when each processor has the same workload (requiring extra memory and time to move data around). In this case, the performance gain from adding more processors is limited by the communication overhead.

AD-A189 844

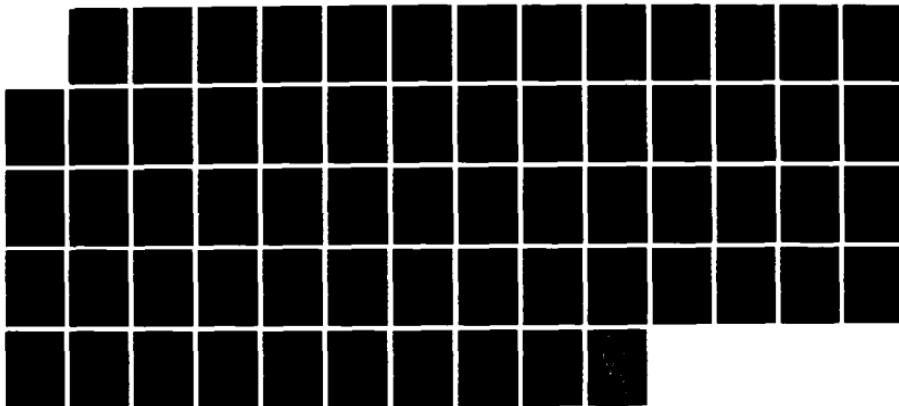
A METHODOLOGY BASED ON ANALYTICAL MODELING FOR THE  
DESIGN OF PARALLEL AND. (U) AIR FORCE INST OF TECH  
WRIGHT-PATTERSON AFB OH SCHOOL OF ENGI. T G KEARNS  
DEC 87 AFIT/DS/ENG/87-1

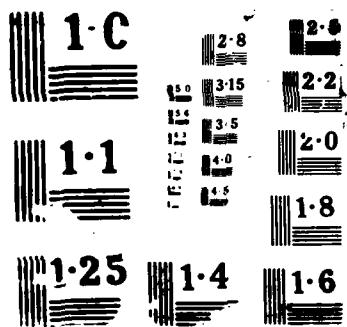
4/4

UNCLASSIFIED

F/G 12/6

NL





each processor.

One solution to the control problem is to have each processor communicate with a central controller at the completion of a task for instructions on what operation and data to operate on next. Some form of distributed control must be considered. since the the single controller would become a bottleneck in the system, processing all the messages for instructions. Therefore, a more definitive control structure for the multi-step query is considered. Any workload adjustment must then be done within the processing environment instead of by a controller.

The first step in task allocation of the multi-step query is the retrieval of the base relations from secondary storage. The options available here are limited by the storage distribution of the relations and the resources that can access the secondary storage. The only situations that seemed feasible were having each relation distributed across all secondary storage devices or storing each relation across one-half of the secondary storage devices (where the minimum amount of a relation stored on a device is one block). The former case provides faster retrievals by using more resources. However, the second case provides the ability to use independent parallelism if the input relations are stored on different devices. If the relations stored on half the disks happen to be stored on the same set of disks, then some of the resources are unused. The results of the multi-step model show that the parallel retrieval of the relations is effective if the inputs are approximately the same size to balance the workload. The complete distribution of the tuples over all storage devices is least affected by varying sizes of inputs and can best accommodate all cases. An example illustrating this uses two relations X and Y, with sizes x and y, respectively. Assuming there are p processors that retrieve relations from p disks. if all p processors are used to retrieve each relation, then each processor retrieves  $(x+y)/p$  blocks. If x and y are stored on different devices, then the workload model is  $x/(p/2)$  for one set of processors and  $y/(p/2)$  for the other processors. When  $x = y$ , this workload is equal to  $(x+y)/p$ . However, if x is twice as large as y, then

the processors retrieving Y have twice as large a workload, destroying the balance of the allocation. Thus, for all cases the complete distribution of relations provides the best capability for retrievals.

The easiest method of data placement to distribute a relation across several disks (using horizontal partitioning of the relation) is a round-robin algorithm [43,42]. The round-robin places equal parts (block level granularity) of the relation on each disk and each additional block added in disk sequence to insure that any one disk has at most one block more than the other disks. This method insures that each disk-processor pair has to retrieve and scan  $1/d$  or  $(1/d)+1$  (where  $d$  is the number of disk-processor pairs available) blocks of the relation. If hardware expansion occurs adding  $x$  disk-processor pairs, then data can be redistributed and the blocks retrieved from each disk is reduced to  $1/(d+x)$  blocks. The round-robin data placement algorithm provides the capability for a linear decrease in performance for a corresponding hardware addition.

The round-robin data placement does not consider the contents or value of data when it is placed. This invalidates the placement of data in ordered format. Therefore to provide the user capability to have the data structures described in the Chapter VII, the round-robin data placement has to be the underlying placement scheme for the data structure. This means that if the data designer determines that a clustering index (indexed-ordered) is necessary to meet the performance requirement for the queries of that relation, then the relation would be ordered and indexed but the data, in ordered form, would be distributed to the disk with equal number of blocks to each disk-processor pair. However, further updates of the indexed-ordered structure may cause one disk to contain an unequal portion of the relation. Then if a query requires a retrieval that does not provide the attribute used for the index, the retrieval would have decreased performance due to the overload of one processor-disk.

The parallelism involved in the retrieval phase of the multi-step query was intranode parallelism versus independent parallelism. For this case, the intranode

parallelism provided the preferred solution because of its ability to adjust for all cases. The next phase of the multi-step query hashes the results of the retrieval phase. The inputs of this phase are not predetermined since the retrieval also performs selection and projection operations. The projection of the attributes can be predetermined since the data dictionary provides the attribute's size. Thus, the compilation of the query could provide this information to allow the controller to partially predict the size after the sel-proj/retrieval phase. The result size cannot be predicted unless distribution information about the attributes used in the selection is maintained and then only approximate predictions could be developed. However, if the controller can identify that the selection criteria uses only the key of the attribute of the relation, then knowing that the key cannot be duplicated and the range of the key provides a prediction of the size of the results.

The purpose of predicting the size of the results of the sel-proj/retrieval operation is to provide the ability to better allocate processors for the next operation. The size of the results of the join and product operations have been previously discussed in relation to the selectivity factor. This showed that the size of the results of these operations could be larger than the sum of the size of the inputs. Therefore, the binary operations, join and product, do not provide enough information to provide a good estimate of the size of the results from the size of the inputs.

The other binary operations do provide a more accurate estimate of the size of the results because the largest size of the results cannot exceed the sum of the sizes of the inputs for union, difference, intersection or division. Further examination of these binary relational operators shows that the difference, intersection, and division operations compare the two input relations and only provide tuples from one relation when the operator condition is met. Therefore, the size of the results from these operations cannot exceed the size of the largest input relation.

The estimation of the size of the results provides the capability to more closely allocate resources to the task using the results as input. Examining the multi-step

query model presented previously, the inputs of the hash/join operation can be estimated. The allocation of processors should match the resources to the problem. For the join, the ideal environment would have each processor joining one block of each relation. However, this causes much of each processor memory to be unused and causes many blocks to be passed from the hashing processors to the join processors. The extra communication is caused because each hash processor must send at least one block to each join processor, even if no tuples are passed to the join processor. Therefore, the immediate goal of resource allocation is to not overload processors, thus avoiding the requirement for data to be stored on secondary storage by individual processors.

The estimate of input size of the input relations of the join processing is determined from the base relation's sizes and the projection factor. Then the processors are allocated so that each processor would receive enough blocks from each relation so that the total number of blocks at each processor is less than the memory size of the processor. If resources are available to allow each processor to process less blocks than this, a minimum number of blocks per processor should be determined. The first consideration must be that the smaller relation has at least one full block at each processor assigned to perform the join. Since the estimation is not precise and the hashing functions will not evenly distribute the buckets, the number of buckets at each processor of the smaller input relation should be 2 or 3. If this forces the sum of input blocks estimated for each processor to be larger than memory available at the processor, the first rule of not overfilling the processor applies and more processors need to be assigned. The purpose of assigning 2 or 3 blocks of the smallest relation to each processor is to eliminate the communication of partial or empty blocks of information. When more processors are added to the join environment, each hashing processor has to send at least one block to each join processor. This could cause delay to the unnecessary communication time. The results of the multi-step query reflect this situation where adding more join processors increases the performance

time rather than reducing it. If both the goals for the minimum and maximum number of blocks allocated to a join processor can be accomplished, then an arbitrary number of blocks desired at each join processor must be established that balances the physical capabilities of the processor and communication network. The multi-step query model provides the capability to vary parameters to the exact physical parameters to establish the desired number of blocks for each given physical case.

The last issue of control to be addressed here is the splitting of tasks among processors. This concerns the splitting of the hash and join functions of the bucket join (and other equal comparison operators that use bucket processing). The results of the multi-step query model presents mixed conclusions about using processors for both actions or using dedicated processors for each action. The results illustrate the fact that causing a bottleneck at any point affects the performance of the entire operation. Therefore, the balance of each step to the amount of information being processed is the most important feature of task allocation. Therefore, the situation that provided the complete base relations as input to the hash/join processing showed better results when all of the processors are used to both hash and join. For situations that reduced the size of the inputs, the resources were more than adequate so the separation of tasks provided better results. This relates to the processing load allocated to each processor described in the previous paragraph. This proves that a task allocation strategy is not the best for each case and the exact allocation algorithm must balance the processor speed and memory capability with the communication time and capability.

### *8.8 Summary of Combined Step Effects*

The ability of the system to perform individual relational operators efficiently is important, but a system that performs a single operator very efficiently may not provide the same level of support when the query depends upon more than one operator to complete the query. The simplest example of a query of this type is the

join. Normally, the query that uses the join does not want the entire results of the join but only selected attributes. This query then not only requires a join operator but also at least a project operation. The concept of query optimization [73,79] also suggests that the input to the join may use a select operation. Therefore, the single join may actually be a query using select, project, and join.

The realization that many queries are actually multi-step queries suggests that some method of combining operators would reduce the necessity of passing data from processor to processor to accomplish each step. The solution to this combined operator was introduced in this chapter. The effect is that data passing is reduced, which reduces the execution time of the query. This method of combining operators also reduces the amount of data in intermediate relations that must be temporarily stored waiting to be used in a later step.

The combination of operators provides the ability to represent any query in a normal form query tree. This general query tree representation promotes the concept of data-flow within the query. Data-flow processing uses the data to control the sequencing of processing (a process is started when the data for that process becomes available) rather than having a constant fixed processing strategy. The normal form query tree uses the idea of combination of operators to reduce the different operations required to solve any query. This reduces the intermediate relations that must be passed (or stored on secondary storage) and the output from each binary node is passed directly to another binary node. This, coupled with the reduced algorithms of the binary nodes, allows the control to be flexible. This means that when a processor finishes comparing one block with another block it must send a message to a central controller to determine what it must do next [12,15,23,62]. Using the normal form query tree combination of operators, this centralized control could be used, but it is just as easy to dedicate processors to the operation where some processors first hash the relations and other relations receive buckets for joining. The recursive application of the hashing allows the workload to be balanced by a processor communicating with

its nearest neighbors to see if they can help in the processing. If the processor cannot get help, it will complete its processing. The performance of the join may be poorer than the optimum performance, but it is still accomplished. By using this flexible control structure, the bottleneck of communicating with a centralized controller is decreased, allowing more processors to be effectively used in processing an operation.

The normal query tree shows the possibility of combining operations. It also shows that there are really only two types of processing necessary to solve any query—the initial node processing and the binary node processing. The initial node processing is the retrieval of the base relation(s) from secondary storage. This initial processing prepares the relations for further processing or solves queries that require only a select and/or project (when no duplicate removal is required after the projection). The binary nodes then complete any combination of relations that may be required or remove duplicates from a relation. The binary nodes may use multiple processors without using the processors that retrieved the data from the disks. This means that when the initial nodes complete the initial retrieval and pass the relation on to the binary nodes, the initial nodes are ready to process another retrieval.

The final conclusion is that the normal form query tree offers possible performance improvement when retrieving data using relational operators. But at the same time, the normal form query tree solves queries with only three different types of processing—sel-proj, bucket-processing, and nested-loop processing. This simplifies the processing and reduces the volume of data passed by combining the individual relational operators into combined operators. These combined operators in the query tree allow all three forms of parallelism to be used in the multiple processor environment. The implication of this is that the database machine has two sets of logical separated tasks to perform in the initial and binary nodes of the normal form query tree and this maps to a two stage architecture for the query processor.

## *IX. Database Machine Architecture*

The architecture of a database machine consists of the hardware, software, and their integration. These components and their integration form a complete environment for solving queries. The focus of this research was not on producing the "ultimate" database design but on producing the tools necessary to complete a database machine design. The architecture presented incorporates the design concepts without defining the actual physical hardware of the system. Thus, the architecture is a logical design of a database machine with suggested hardware implementations. However, the models presented provide the capability to complete the design by applying the trade-offs of cost versus performance to complete the physical design. The design considerations present a general environment, but the models presented allow the design to emphasize one system requirement and determine the effect of this on the performance on other operations. The following sections discuss the application of the performance models in the design of a database machine. First, a logical architecture is presented and then the architecture is discussed, providing possible physical implementations for each logical segment of the logical architecture.

The database machine is designed under a given set of guidelines or assumptions. The first step in presenting the database architecture is reviewing the guidelines used to develop the architecture. The basic assumptions are:

- The database machine is capable of handling any size database. Thus, the size of the relations may exceed the total memory size of all the processor(s).
- The processors form a MIMD environment with each processor having the ability to communicate with other processors.
- Each processing unit is a generic type processing unit with processor, memory, and the ability to execute a stored program.

- The focus is on improving the performance of data retrievals and updates (backup, checkpoints, and recovery are not considered).
- A data dictionary must be maintained identifying the relations in the database and the properties of the relations. This is considered to be a separate entity and will not be considered as a part of the database machine architecture. The database machine will receive queries containing the necessary data definitions included in the query.
- The performance optimization is accomplished through the use of parallel processing and effective utilization of resources. Optimization techniques of re-ordering the query steps are not considered to be a part of the database machine architecture but part of the processing of the query before it is given to the database machine.
- Expansion of the system must be possible to handle larger workloads.
- The database machine can handle any relational algebra equivalent query. All retrievals will be considered in terms of relational operators.
- The design of a database machine can be accomplished in a structured method by examining the problems to be solved, analyzing the individual steps of the problem, and using the analytical models for each step of the problem to support the trade-offs decisions necessary to create an integrated database machine.
- Queries to the database machine are formed outside the database machine and transferred to the database machine. Also, the output from the database machine is considered to go to a data sink capable of accepting data as fast as database machine can produce it. Therefore, the solution of a query is not assumed to be constrained by the speed of the connection removing the data from the database machine. This does not imply that the database machine has to be a back-end machine to another computer, but that part of the sys-

tem supporting these functions could be part of the same physical computer.

Figure 54 shows the logical environment of the database machine.

The design of the database machine has been based upon a structured approach to the analysis of the problem of retrieving the data to satisfy a query. The following sections present a logical architecture based upon the analysis presented in the previous chapter. This abstract architecture is then expanded to one possible physical architecture. This exercise is intended to illustrate an methodology for using the models developed in this research and is not intended to be an "optimal" architecture itself.

### *9.1 Database Machine Architecture*

The database machine has two types of operations to process—simple retrievals (data filtering) or the combination of two relations in a binary operation. These two types of processing are illustrated by the normal form query tree with its initial and binary nodes (see Figure 55). This defines a two stage or two part architecture for the database machine. Figure 56 shows the logical database machine architecture suggested by the normal form query tree. The retrieval stage corresponds to the initial nodes and the binary nodes are the processing stage.

Each stage has a given set of tasks to process. The retrieval stage retrieves relations from secondary storage. The results of this stage are then passed to the processing stage or are output. This corresponds to the initial nodes of the normal form query tree. The processing stage performs the actions of the binary nodes. The data flow of the processing stage retains the intermediate results within the processing stage and outputs the results to the "host". Figure 57 shows the logical flow of data and separation of tasks of the database machine.

*9.1.1 Retrieval Layer.* The main task of the retrieval stage is the retrieval and filtering of the base relations for desired tuples and attributes. The results of

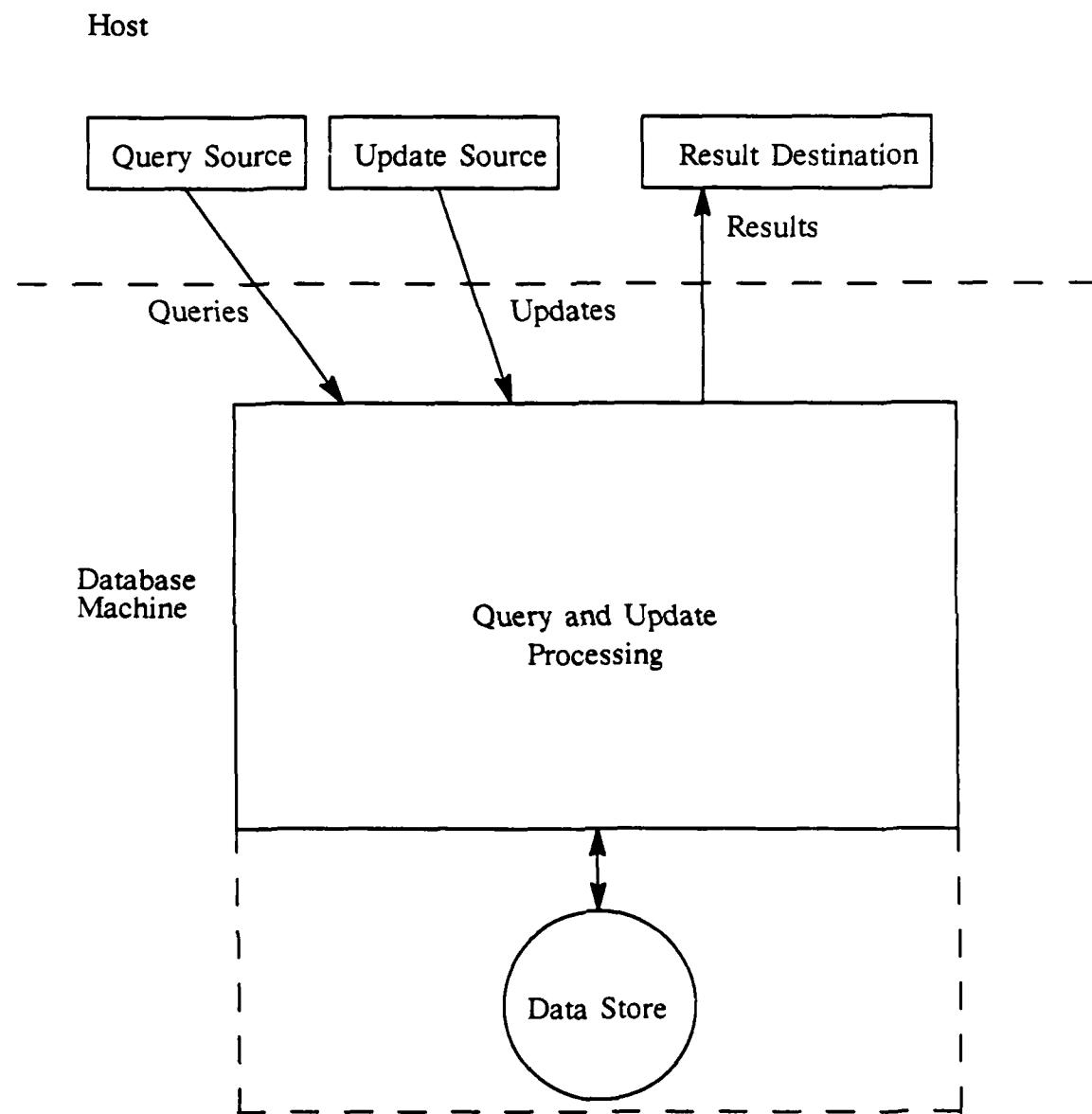


Figure 54. The Logical Database Machine Environment

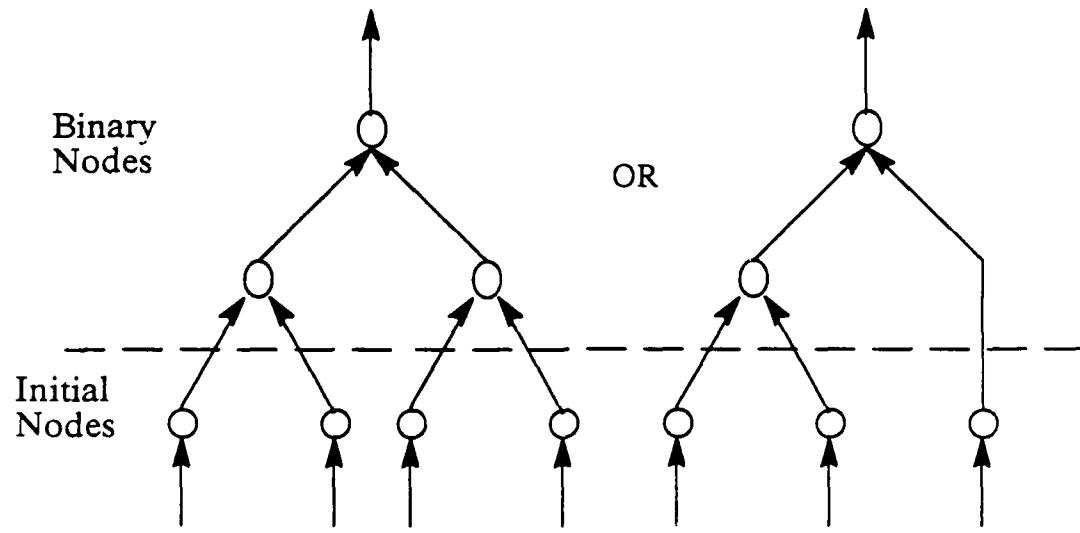


Figure 55. General Normal Form Query Tree

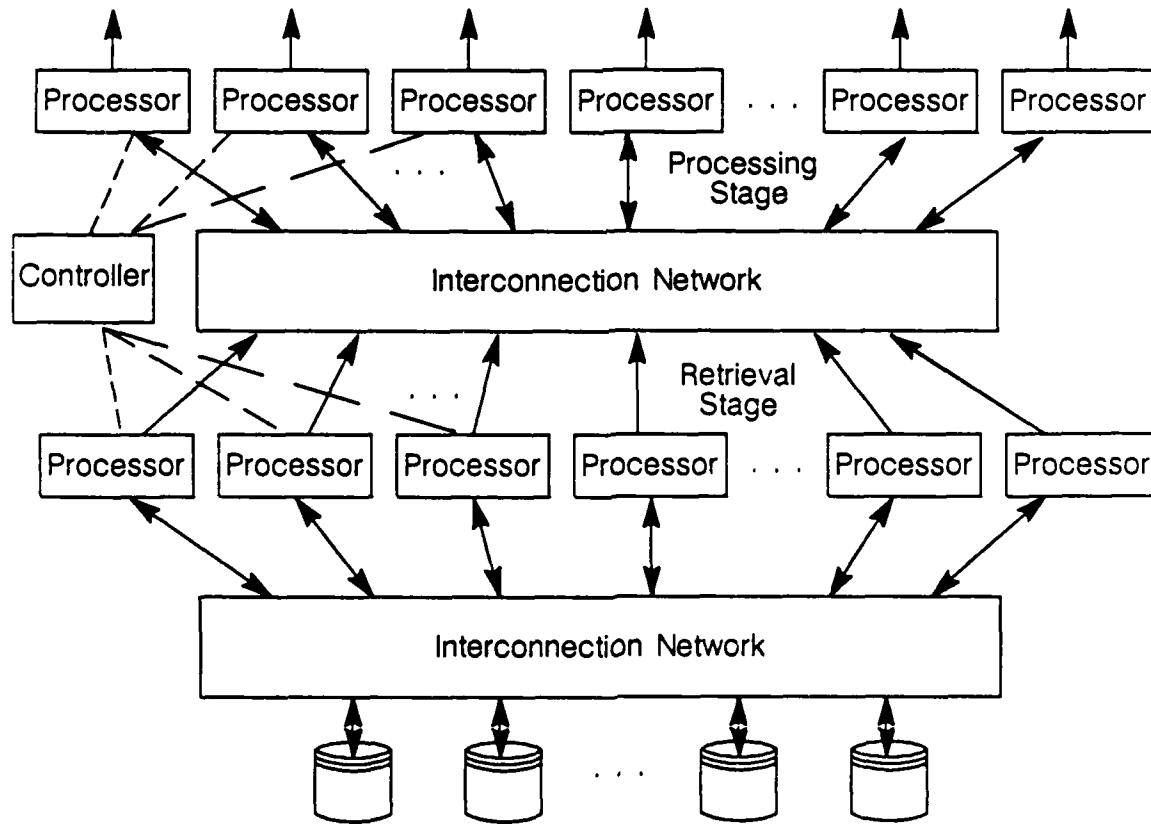


Figure 56. Logical Database Machine Architecture

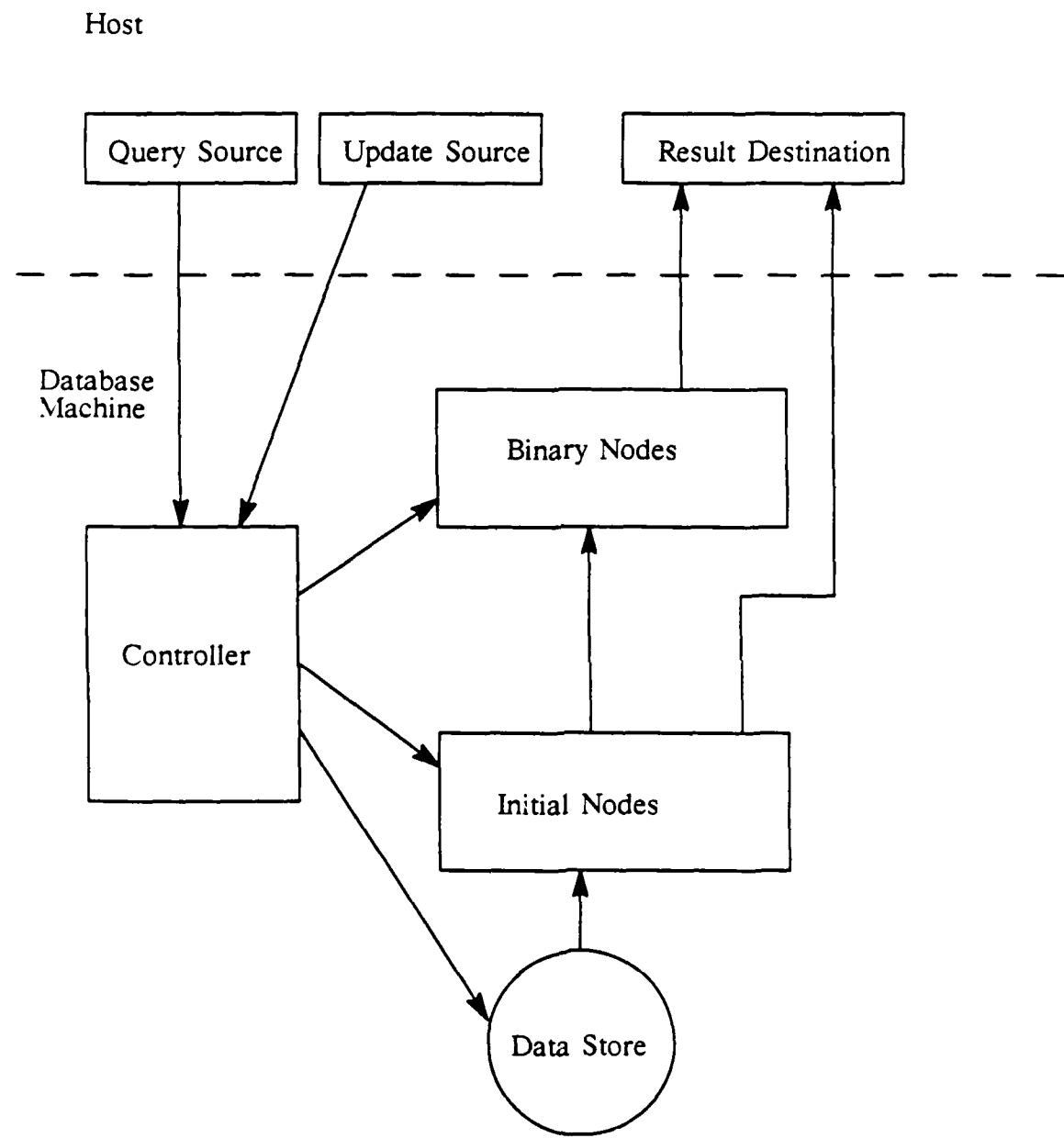


Figure 57. Logical Data Flow of Database Machine

the retrieval stage can then either be returned (if this completes the query) or passed to the processing stage for further processing. The other task of the retrieval stage is to complete updates to the database.

The only retrieval operation executed by the retrieval stage is the sel-proj operation. Each processor executes the sel-proj and passes the result on to the processing stage or to be output to the user. If the entire relation is desired, the sel-proj selects all of the tuples of the relation and removes no attributes.

The tasks of the retrieval stage may seem very limited. The lack of the retrieval stage participating in further processing of queries beyond initial retrieval and filtering was intentional. The purpose of this is two-fold: to provide the capability to reduce the I/O bottleneck [11,13] when accessing the database and to improve the throughput of the system [11] by allowing simple query retrievals to be completed while the processing stage is executing more complex queries. The only method possible to insure this for all retrieval cases is to use parallel processing and distributed data storage. The use of parallel processing does not provide the best retrieval in all cases (see Chapter VII), but even optimized techniques using indexing may not have an index for the attribute being retrieved. Thus, the alternative is to efficiently use multiple processors to retrieve and scan pieces of the relation (reference Chapter VIII).

The other consideration of eliminating the I/O bottleneck is avoiding the reverse flow of data within the system. Reverse data flow occurs when a relation(s) is being retrieved and processed, but the same disks are simultaneously being used to store intermediate results. This causes a conflict within the disk processing, forcing the disk to perform costly accesses to find the correct location on the disk for the data currently being handled. If reverse data flow is eliminated, the disk is only reading data (except for update operations). This allows the disk head to maintain its position, reducing the disk accessing time. Also, improved disk retrieval techniques (interleaved blocks, etc.) can be exploited by avoiding the conflict of disk

accessing. The logical data flow represented in Figure 57. eliminates any reverse dataflow within the system by requiring the processing stage to handle its own temporary storage requirements. This allows the flow of data to go from retrieval stage to processing stage to output. The one exception to this data flow is the handling of updates by the retrieval stage.

The final action of the retrieval stage is to perform updates. All updates consist of a select followed by the modification of a block(s) (reference Chapter VI). During the select, the processors must determine if a tuple(s) was found that met the update condition (or integrity check condition). Finding a duplicate tuple invalidates the update. This forces the processor to notify the controller to stop the update action. If all processors provide the controller positive responses that no duplicates were found, then the controller directs an individual processor to add the new tuple.

The deletion case requires the processor to find the necessary tuple to be deleted and then notify the controller. This allows the controller to maintain (or pass the information to the process that maintains) the data distribution statistics. The tuple is deleted from the block and the processor replaces the block on disk. Therefore, the modification operation requires both a select and delete operation. The action by the processors combines the selection check for duplicates and the finding the tuple to be deleted as the deletion. These cases do not produce results other than a completion message. This allows the retrieval layer to accomplish this action without involving the processing stage.

In summary, the retrieval stage performs retrievals and updates of relations. The controller receives a query, procures any necessary data definition information, and broadcasts the query to the retrieval processors. The processor-disk pairs retrieve their portion of the relation, each processor performs the sel-proj operation and either outputs the results or send the results to the processing stage. The retrieval stage is then available to perform another retrieval or to execute an update.

*9.1.2 Processing Layer.* The processing stage corresponds to the binary nodes of the normal form query tree. The task of this stage is to perform the operations necessary to complete queries that require more than the sel-proj operation. This means that this stage executes join, product, intersection, difference, union, select, project, and duplicate removal operations.

The processing stage performs either bucket or nested-loop processing. Bucket processing requires the relation(s) being processed to be grouped in disjoint sets of tuples and the disjoint sets or buckets are distributed to the participating processors. The processors then order the buckets and perform a join, a difference, an union, an intersection, or duplicate removal. To perform a bucket process, the ideal number of processors involved would be enough processors to hold all blocks of the input relations in the memory of the processors with an evenly distributed set of buckets (see previous chapter). In an actual distribution, one bucket maybe larger than all the others. This could require the processors to handle more data than could be contained in the processor memory. This requires some form of secondary storage to be available to the processing stage.

The secondary storage required by the processing stage could be provided by the storage associated with the retrieval stage. However, using this storage destroys the one-way flow of data that has been established. Therefore, Figure 58 shows the modified logical database machine architecture to include some temporary storage capability at the processing stage.

The processing stage consists of independent processors with communication capability. Based on the results presented in an earlier chapter, it is assumed that the communication network provides a fully interconnected processor communication network. This allows any processor in the processing stage to communicate directly with any other processor in the processing stage. The advantage of this communication with the bucket processing is that an overloaded processor can request help from neighbor processors and hash the buckets it received to redistribute the workload.

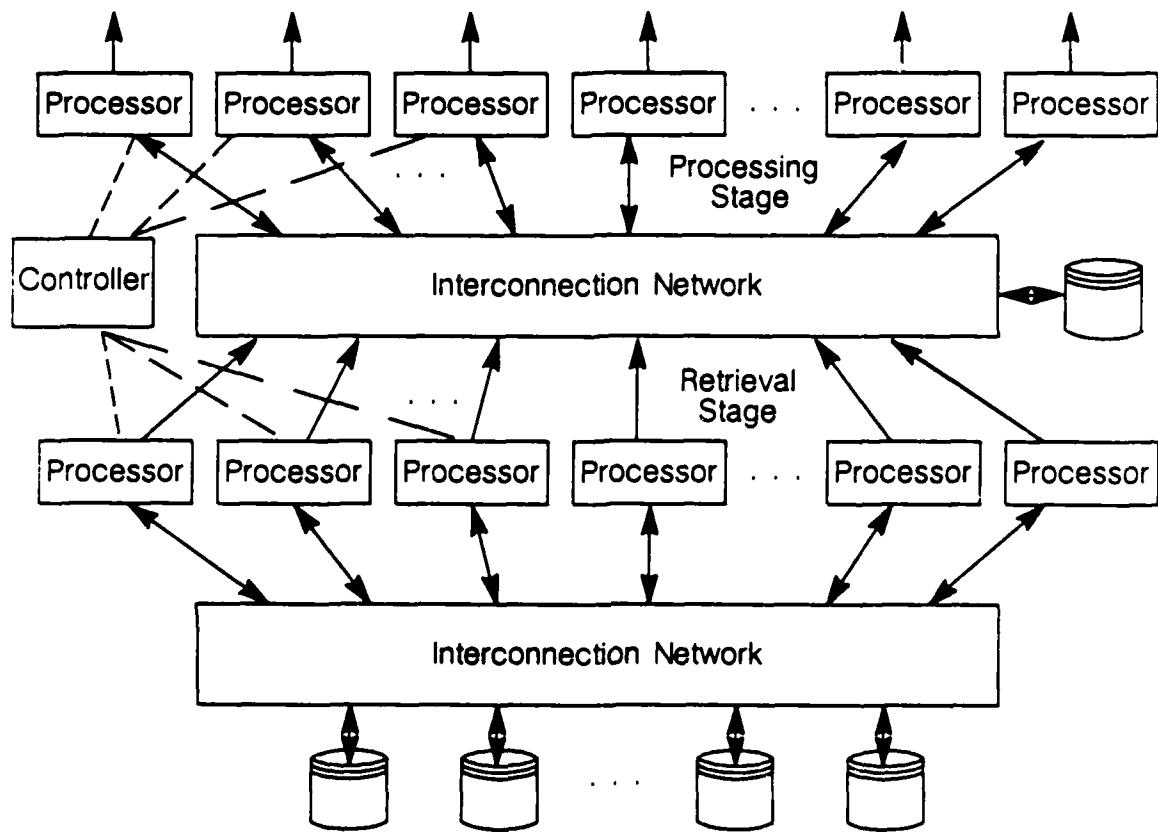


Figure 58. Modified Logical Database Machine Architecture

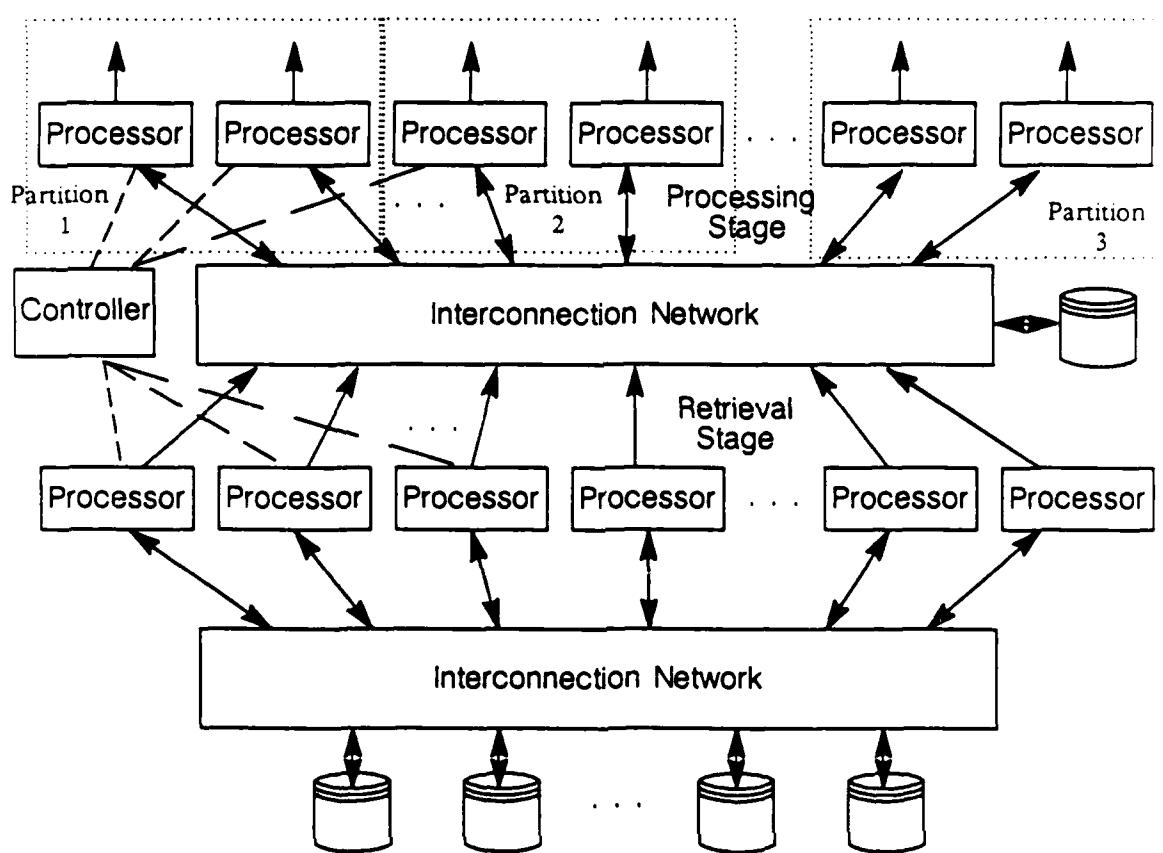


Figure 59. Logical Database Machine Architecture

This recursive application of the bucket processing provides a dynamic adjustment to the problem of task allocation.

The processing stage reflects the binary nodes of the normal form query tree. The binary nodes of the query tree perform the bucket processing described above and also require a nested-loop process. The normal form query tree also shows that there are multiple nodes to execute at a given level of the query tree. The logical view of this is that the processing stage should provide the capability to perform one or more two different operations simultaneously. This alters the control view of the processing stage to provide the ability to partition processors of the processing stage into separate control groups. Figure 59 shows this final logical view of the database machine architecture.

The partitioning of processors allows one set of processors to execute a nested-loop process while another partition is executing a bucket process. The partitioning of processors also provides the capability to simultaneously execute multiple queries. The partitioning is a logical grouping of processors. Therefore, the partitions are not fixed groupings but an assignment for an individual task.

The partitioning of the processing stage provides the ability to adjust the resource allocation to the problem. Partitioning also allows a decentralized control system. A single overall allocation scheme would perform the partitioning or resource allocation of processors, but a local processor of each partition would then assume control of the processing within that partition. This provides a flexible, adaptable processing configuration.

## *9.2 A Database Machine Design*

The logical architecture defines the function of each logical component of the database architecture. In the final step, the logical components need to be mapped to a physical architecture. The following sections describe some of the design and performance considerations in mapping the logical database architecture to a physical implementation. One possible physical implementation is illustrated in Figure 60.

The design of a database machine consists of the physical mapping of processing requirements to hardware, but also incorporates the software to control the processing. In addition, the physical structure used to store the relations must be defined. The goal of the design is to provide the best performance in solving relational queries. However, every design has some limitation due to the physical characteristics of the hardware, lack of knowledge in the software control, and/or cost restrictions. The following sections review the logical tasks to be performed by a database machine and the use of the analytical models in the design of a physical implementation of the logical database machine architecture.

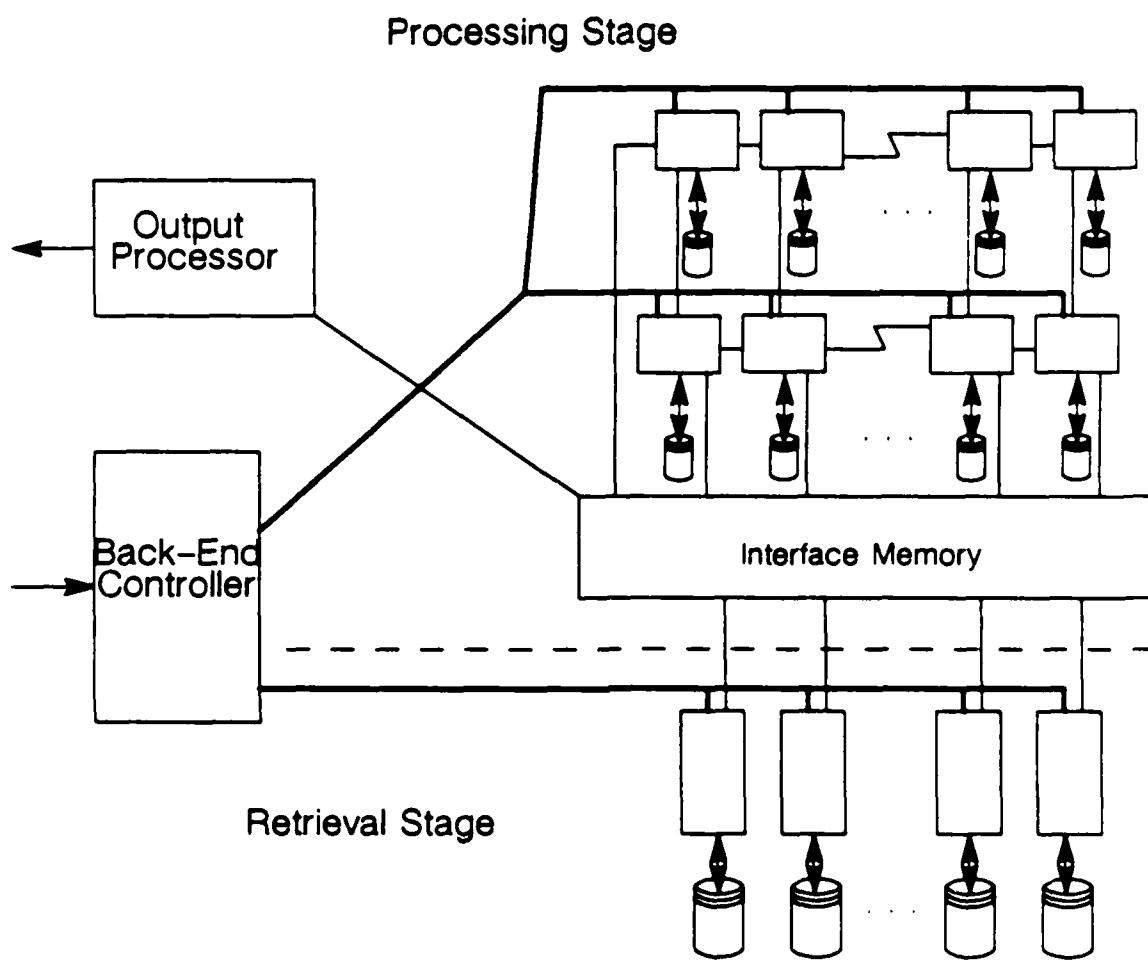


Figure 60. A Database Architecture

*9.2.1 Retrieval Stage.* The retrieval stage is responsible for maintaining the "permanent" database and retrieving relations from this database to be used to solve user queries. The speed of execution is the main consideration of the retrieval stage. The analytical analysis of retrieval operations has determined that there are different methods of optimizing the retrievals (Chapter VII) depending upon the data structure of the stored relations. The conclusion of the analytical models was that optimized data structures could provide optimized retrievals or the same performance could be obtained using an unordered-unindexed data structure with more physical hardware for retrieving and storing the data. However, for every case it was determined that the relation should be distributed for storage. This distributed storage pattern of the relation maintains an equal number of blocks on each disk (where each disk holds a horizontal fragment of the relation). Also, the modeling showed that multiple processors accessing a single disk create a contention problem that increases the performance time of a retrieval. Therefore, the goal of a design would be to associate a single processor with each disk or set of disks. The mapping of the retrieval stage functions to hardware requires physical hardware to store and retrieve the relations. The situation desired by the mapping is to associate a processor with a disk as described above. However, cost restrictions may make this infeasible or reduce the number of disk/processor pairs from the desired performance level. This provides the opportunity of using the analytical models to evaluate the different options available and to determine the best parameters to meet the constraint. An example of using the analytical models might compare the performance of retrievals when using 5 disk/processor pairs versus using 3 disks with 10 processors. Also, the models provide the ability to compare the various data structures with the different configurations to determine if this may provide the desired performance level within the cost constraints.

The other design consideration of the retrieval layer is the update capability. For some environments, the performance of updates may be as critical to the total

system performance as data retrievals. Using the update and retrieval models, the system can be balanced to provide a design to give the desired performance. The architecture presented mapped the retrieval stage functions to a set of disk/processor pairs with the relations distributed over all the disks.

The advantage of the round-robin distribution of the data and the disk pairs is that adding new disk/processor pairs and redistributing the data can provide an linear performance improvement. Since each disk may contain a portion of the relation, each processor/disk participates in the retrieval or update of a relation. Therefore, the retrieval stage is a SIMD environment.

The SIMD environment of the retrieval stage suggests that the controller have the ability to broadcast the retrieval or update instruction to each processor. Also, during the update operations the processors needed to communicate with the controller. This means that each processor must have a communication line to the controller. The physical implementation of this could be individual communication links or a common bus.

*9.2.2 Retrieval-Processing Stage Interconnection.* The processors of the retrieval stage pass the results of the sel-proj operation to the processing stage for further processing in some cases. The logical view of this data transfer and outputting of results is that the retrieval processors have a full interconnection to each processing stage processor. The mapping of this logical communication capability to physical implementations is limited to some form of network or physical connections between processors. Since the time required to complete this communication affects the performance of the entire query retrieval, the analytic models provided the capability to change the communication time parameter to model different physical implementations and the performance effects of each.

The obvious physical implementation is a full interconnection network for passing the results between stages. However, if a processing stage processor is to receive

the results from a retrieval processor and the communication buffers of the processing processor are full, the retrieval processor must wait until buffer space becomes available. Also, the assumptions stated that the database should be adaptable for expansion. The fully interconnected network requires each addition to be connected to every other processor. This may not be adaptable for expansion. Therefore, the suggested implementation of the information exchange facility between the retrieval and processing stage includes an interface (cache) memory.

The advantage using an interface memory consisting of cache memory is that the controller tells a retrieval processor which block to pass data to and the processor does not have to wait for a processor to receive the data. Also, the cache memory provides a broadcast type ability by allowing several processors to read the same block of data. The cache memory requires an interchange network between each stage and the memory. But expanding either the processing or retrieval stages has no effect on the other stage because of the modularity provided by the interface memory.

*9.2.3 Processing Stage.* The processing stage corresponds to the binary nodes of the normal form query tree. This requires the processing stage to perform join, product, intersection, difference, union, and duplicate removal operations. The processing stage accomplishes these operations by using two types of combined operator processes: nested-loop and bucket.

The bucket processing uses the equal-comparison type processing (Chapter IX). Included in the equal-comparison process is the removal of duplicate tuples from each input relation. Therefore, if the final process concludes with a projection that may introduce duplicate tuples, the results would have to have the duplicates removed. For all the processing, the logical architecture requires the processors to have the ability to pass information to other processors. All of the performance modeling results presented in previous chapters assumed no communication restrictions.

This means that the implementation to provide this would be a full interconnection communication network connecting the processors.

A fully interconnected set of processors is suggested as the mapping of the logical architecture to a physical architecture. However, this may be too costly or physically impractical to implement. Therefore, the design process must determine the communication capability/number of processors ratio that best fits the requirements of the system. The models developed provide the capability to analyze these different situations to predict the performance capability of each. Especially critical is the consideration of the multi-step queries if this type processing constitutes much of the workload of the system.

A full interconnection network would be the best implementation from the performance view. From a practical implementation standpoint, the full interconnection network may not be feasible. The control structure of the logical database architecture suggested that the processing stage be logically divided for control into partitions. The partitioning of the processing stage provides a logical division of the communication required within the processing stage. This logical division allows the communication between partitions to more fixed points. In current multiprocessor environments, the hypercube interconnection structure is an alternative interconnection structure [78] to a fully interconnect network. Therefore, the hypercube interconnection structure will be used to show how the interconnection network may constrain the processing and how the processes can be modified to better utilize the network.

The task allocation for the best performance suggested that for bucket processes each assigned processor hash part of the relation and then all processors participate in the join or comparison processing. Also discussed was a task allocation that assigned some processors to only hash and others to only process the buckets. The concern of the dedicated task allocation was maintaining balance and not forcing a small number of processors to do an inordinate amount of the processing.

The limited communication of a hypercube (limited direct communication between processors without using intermediate nodes) does not support the full interchange required by the shared processing control structure (without time delays of passing the data through intermediate nodes). A suggested modification to accommodate the more limited communication capability is to use a modified dedicated task allocation.

The modified task allocation to perform the bucket processes assigns some processors to hash portions of the inputs, retaining a portion of the input for later processing. The hashing processors only hash the input into small number of buckets. The processors receiving the first level buckets now hash their inputs, retaining a portion of the buckets for later processing. The second level inputs already form logical groups so the hash of the second level is just breaking the groupings into smaller pieces. Thus, the results of the second level do not have to be fully interchanged. Figure 61 shows the logical operation of this modified hash/join process. The figure only shows 3 levels of processing. However, the processing of level 2 could be replicated at each level to provide the fanout of data desired. The constraint of this method is that the lower levels must scan a larger portion of the inputs. But the purpose of the modified bucket process was to adapt the process to limited communication capability of the processors.

The other operation of the processing stage is the nested-loop. The nested-loop operation compares each tuple of the inputs. The nested-loop operation is discussed in Chapter III. The control situation of the nested-loop is that operation is assigned to a partition(s) and a processor within the partition is given the task of controlling the process.

The operations of the nested-loop or bucket processes either provide the results to output to the host or temporary relations to be used as input into another binary node. If the results are input into another binary node, the controller directs the results to go to another partition for processing, or the results could be sent to

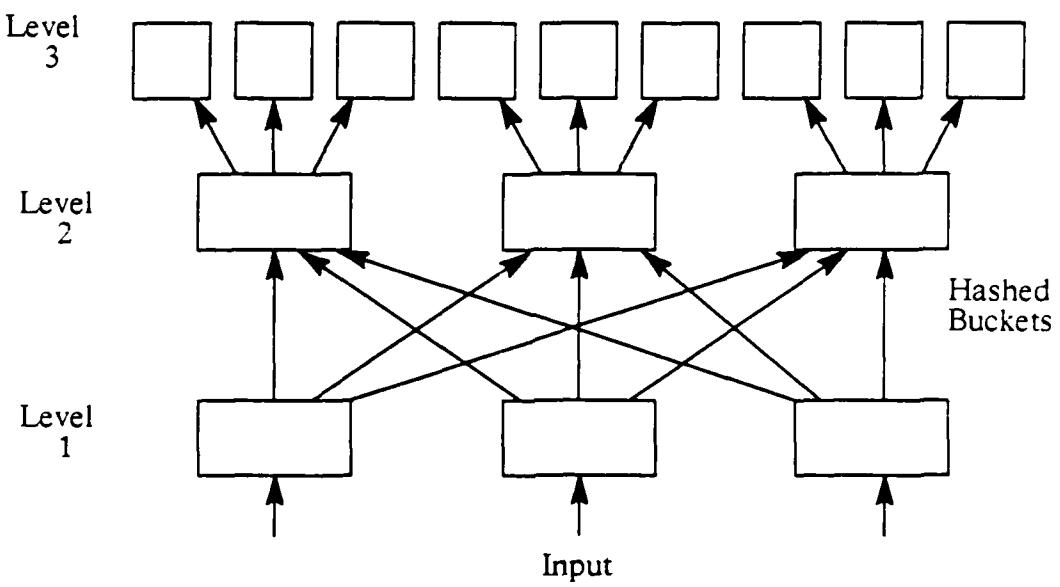


Figure 61. Logical Operation of Modified Hash/Join Process

the interface memory to be processed. If the results are output, a contention could develop. The output in all the analytical models was assumed to go to a parallel data sink. To implement this would require parallel lines connecting the "host" with the processing layer. This may not be feasible. Therefore, an implementable method of simulating parallel output is to send the results to a segment of the interface memory. Then an output processor can retrieve the results from the interface memory.

The operations discussed above have all tried to effectively use the resources available to eliminate having to store temporary results on secondary storage. However to be able to handle all cases, this is not feasible. This requires the processing stage to have some secondary storage capability. The implementation of this secondary storage capability needs to maintain the dataflow within the system. This means that the retrieval stage is not considered to be a temporary storage depository. The secondary storage suggested by Figure 60 is some disk storage at each processor. This ability to access secondary storage without contention from other processors is the assumption used in forming the performance models of multi-step query. An

alternative but less desirable method would provide only one disk for each partition. This would provide the storage capability but could increase the performance time of an operation that required secondary storage to be used because of the contention time in accessing the disk.

**9.2.4 Controller.** The controller is the interface between the "host" and the retrieval and processing stages. The controller directs which query or update to process and provides the necessary information to perform the operation. The task allocation techniques for individual queries has been discussed in the previous sections and previous chapters. However, one other factor to improve the throughput of the system is the sequencing of query execution.

Sequencing the query execution can improve the throughput of the system by utilizing all parts of the system. The best example of the controller sequencing queries is when the database machine needs to process a multi-step query and also a simple retrieval query. If the multi-step begins first, there reaches a point where the retrieval stage completes the retrieval of the base relations. The retrieval processors are then available to perform the retrieval query which only requires action by the retrieval stage. Also, updates could be accomplished by the retrieval stage while the processing stage operates. This provides the capability to improve the throughput of the system by separation of tasks.

The result is that by using a modular design of a database machine based upon the analytical analysis of the relational operations, the capability to improve throughput exists. This modularity of the system also allows the capability to expand the system to improve the performance as needed.

The modeling provides the basis, then, for developing a physical design by predicting the capability of a given situation. This reduces the design considerations in mapping the requirements of the system to a design. The general form of the query tree also aids the design by logically grouping the processing requirements of

a query. This provides the capability to model the query performance for various task allocation concepts to help determine the control policy for the system.

## *X. Conclusions and Recommendations*

### *10.1 Conclusions*

The goal of this research was to provide the capability to design, using a structures design approach, an improved database query processor through the use of a multiprocessor environment. The use of a multiprocessor environment suggested the use of parallel processing to improve the data retrievals. To avoid using intuition as the guide for the design of the system, a thorough examination of the feasibility of parallel processing of relational operators needed to be done. This examination explored the theoretical considerations of parallel processing and modeled the performance of different implementations and environments for parallel processing of relational operators. The following discussion summarizes the results of each step in developing the capability to design an improved database management environment.

The first consideration of using parallelism in a database retrieval is determining the theoretical constraints of parallelism in database retrievals. The application of Ullman's properties of relational operator manipulation [79] and set theory principles provided the capability to determine the feasibility of using partitioned relations and parallel processing to execute the relational operators. Two additional properties of simplifying manipulating relational operators, commuting a product with a union and commuting a join with a union, were introduced to facilitate the theoretical considerations. The result was that horizontally partitioned relations provided the least constrained application of parallel processing of the relational operators.

The next step in the design development was determining how individual retrieval operations should be implemented. This incorporated the use of analytical modeling of the individual relational operators. Analytical modeling of relational operators is not new. However, the analytical models presented extended the modeling to include architectural combinations (i.e., sending results to a backend as well as

storing results on disk, using multiple processors with single disk path, and a single processor with access to multiple disks) not previously considered for all individual operators. Also, the models varied the destination of the results to illustrate the impact of the disk accessing time when temporary results must be stored on disk.

The remaining action of the database is updating the relations. The effects of performing updates is usually neglected in the design of database retrievals. However, new demands for immediate, accurate data require the application of real-time updates. The updating of relations includes determining if the update will not corrupt the database (i.e., add duplicate tuples to the database). To provide comparison of the implementation of updates, the analytical models of the retrieval operators were extended to include three update actions: insertion, deletion, and modification.

The 237 query performance models provide the capability to evaluate the performance of the individual query operator implementations for various data structures and hardware environments. Performance results for the individual operators, for a given set of hardware performance parameters, presented conflicting results. Parallel processing was the best method of performing projects and joins but an optimized indexed structure was the best method of implementing the select and update operations. However, the optimized data structures could also allow the underlying tuples to be distributed for storage to allow the multiprocessing environment for all cases to be possible. Thus, the performance of each relational operator may be improved through the use of parallel processing and the performance improvement of additional hardware resources to provide improved parallel processing can be determined by the analytical query models.

The analytical modeling of the individual query steps provided insight into the data structure and implementation of single-step queries. However, it failed to address the requirement that many database queries require a combination of individual relational operations to complete the query. Therefore, the final modeling consideration was to incorporate the multi-step query.

Queries of a relational database can be expressed in a tree form, called a query tree. This query tree provides a dataflow model of the individual query. This dataflow model provided the impetus to define a general model of any database query - a general normal form query tree. The normal form definition of any query is possible by the combination of binary and unary relational operators as a single operation.

The normal form query tree definition was extended to model multi-step queries, providing performance modeling capability for any given query. The general modeling capability used the recursive definition of a binary tree and the combined operation properties of the normal form query tree to provide a recursive normal form query tree. This provides modeling capability for any complex multi-step query.

The analytical modeling of multi-step queries involved the determination of the affect of the data distribution, the hardware architecture, and the control or task allocation of the processing, on the performance of the query. The analytical models developed provide the capability to vary each of these performance variables to evaluate the effects on the performance of executing a complex query. For a given set of hardware parameters, various combinations of data distributions, resources and control structures were used to construct a set of curves depicting the performance trends of the interaction of data distribution, resources available and how the resources are applied to the problem. These results provide guidance for further research in control structures and task allocation.

The building blocks for the development of a database design are provided by the analytical models and general form of the query tree. However, the final step presented showed how the analytical models and normal query form could be used to develop a design. Using the results of the modeling, a logical architecture of a multiprocessor retrieval environment was presented.

The logical architecture incorporates parallelism, intranode, independent, and pipelining, to improve performance. There are two measures of performance - the

time required to execute an individual query and the throughput of the system. The throughput of the system depends upon the time to perform each individual query but also depends upon the capability to provide support for more than one task at a time. The logical database architecture's logical separation of tasks provides modularity and flexibility, allowing a one-way data-flow within the primary parts of the system to reduce the performance delays due to disk accessing contention delays. This provides the capability for improved throughput in the system because the retrieval stage supports the processing stage but it also operates independently. Thus, allowing the queries requiring simple data retrievals to process while more complex queries are being completed, improving the throughput of the system.

The other performance measure, the performance of an individual query, was directly compared to an existing benchmark to show the performance improvement provided by the logical architecture. A benchmark of existing database management systems and database base machines [9] provided the means of comparison for several complex queries. Using the parameters provided from this benchmark, the performance improvement possible utilizing the parallelism concepts developed was shown. The results (shown in detail in Appendix C) showed a significant performance improvement (a minimum of order 5 decrease) over existing database systems and database machines. This shows the potential of a database query processor, constructed using a structured design approach, to provide improved performance over database machine designs that used the intuition and experience design approach.

## *10.2 Recommendations*

The results show the potential for further research to develop an actual implementation of a multiprocessor database machine using the analytic model developed here. Therefore, there are several areas of further research required for improving the design of a database machine. The first research area is actually continuing the design process to implement the processes modeled by the analyti-

cal models. This would provide the capability to perform empirical analysis. This process would also explore the physical limitation presented by physical hardware.

Another area of the database machine design that needs more research is optimizing task allocation. The multi-step query models provide the capability to evaluate the performance of a given query for different control approaches. However, the analytical models must use estimates of the selectivity factors to determine the performance of the given query. The process that allocates the tasks within the system does not have this prior knowledge of the size of the inputs into each node of the query. If the controller had complete knowledge of the size of the inputs at each node, an optimal task allocation could be done (but possibly not in real-time). This implies the need for two different research focuses – size estimation within multiple step queries and task allocation and load balancing in a dynamic multiprocessing environment. The normal form query tree provides an environment that provides the basis for both of these research topics.

## Appendix A. *Multi-Step Query Results*

The following graphs represent the results of the performance modeling of the multi-step query models varying the input sizes, number of processors, and allocation of tasks to processors.

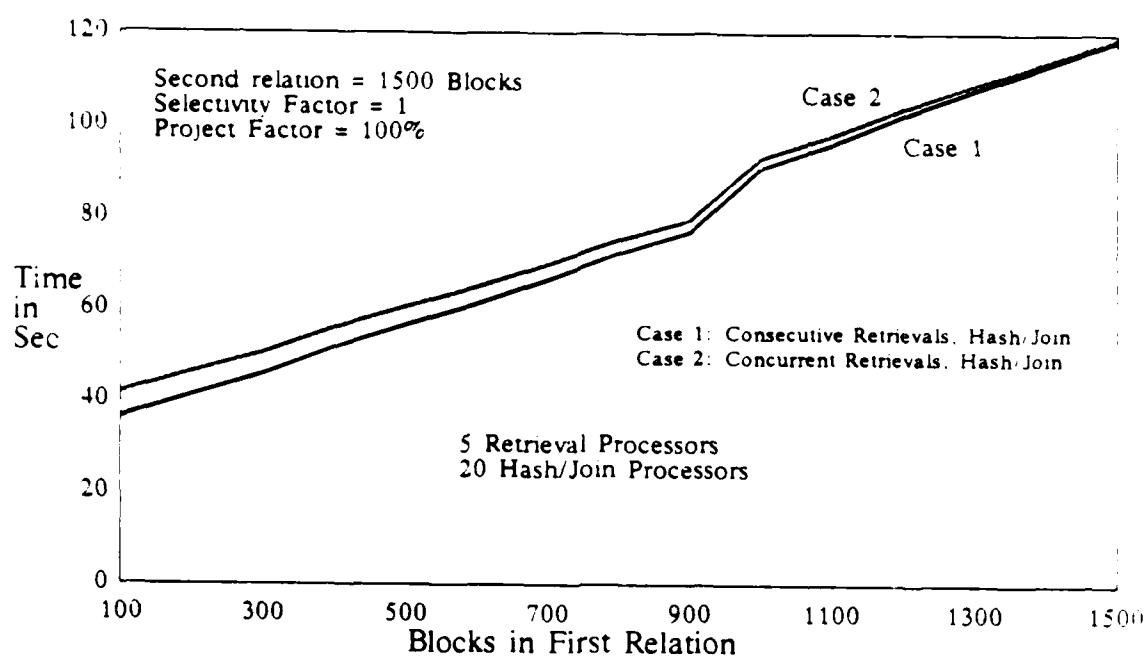


Figure 62. Multi-Step Performance using Hash/Join

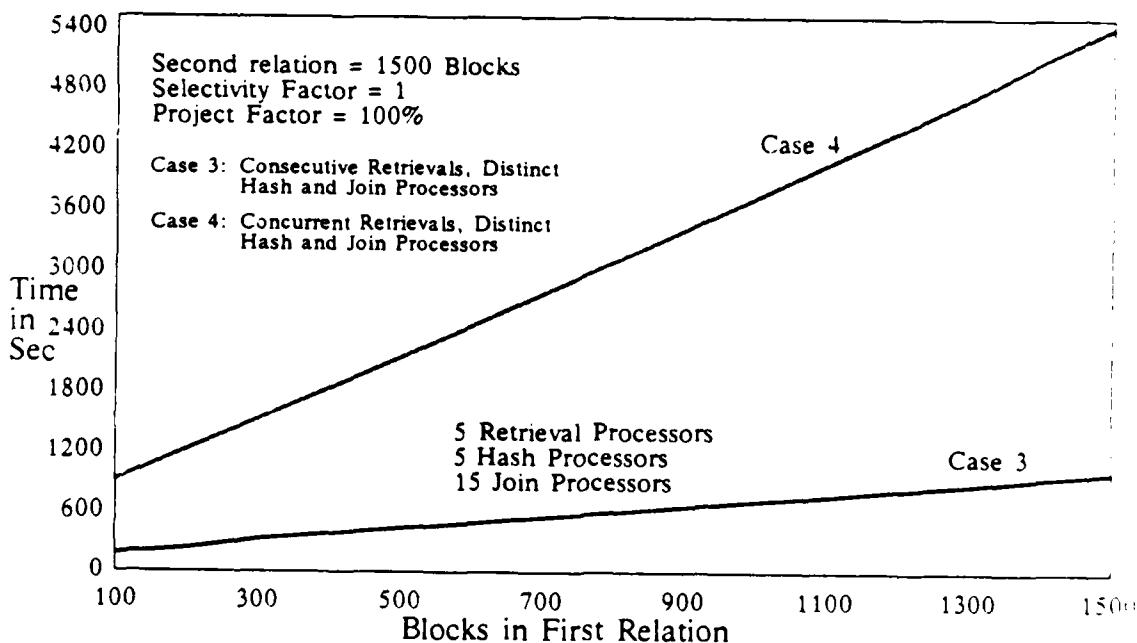


Figure 63. Multi-Step Performance using Separate Hash and Join - 1

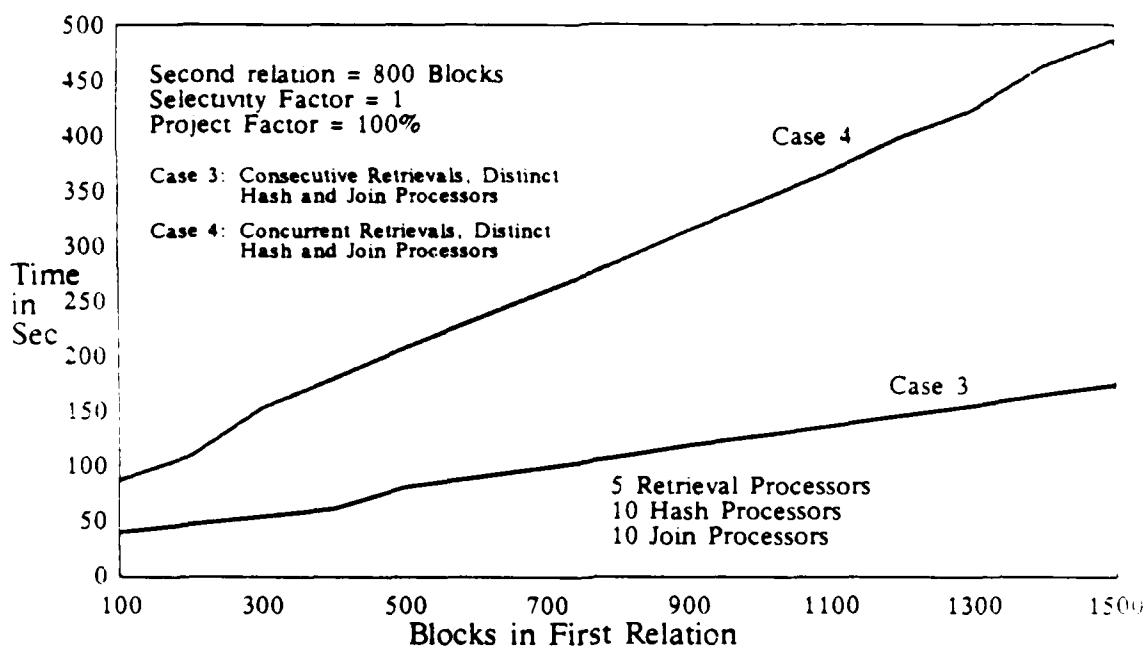


Figure 64. Multi-Step Performance using Separate Hash and Join - 2

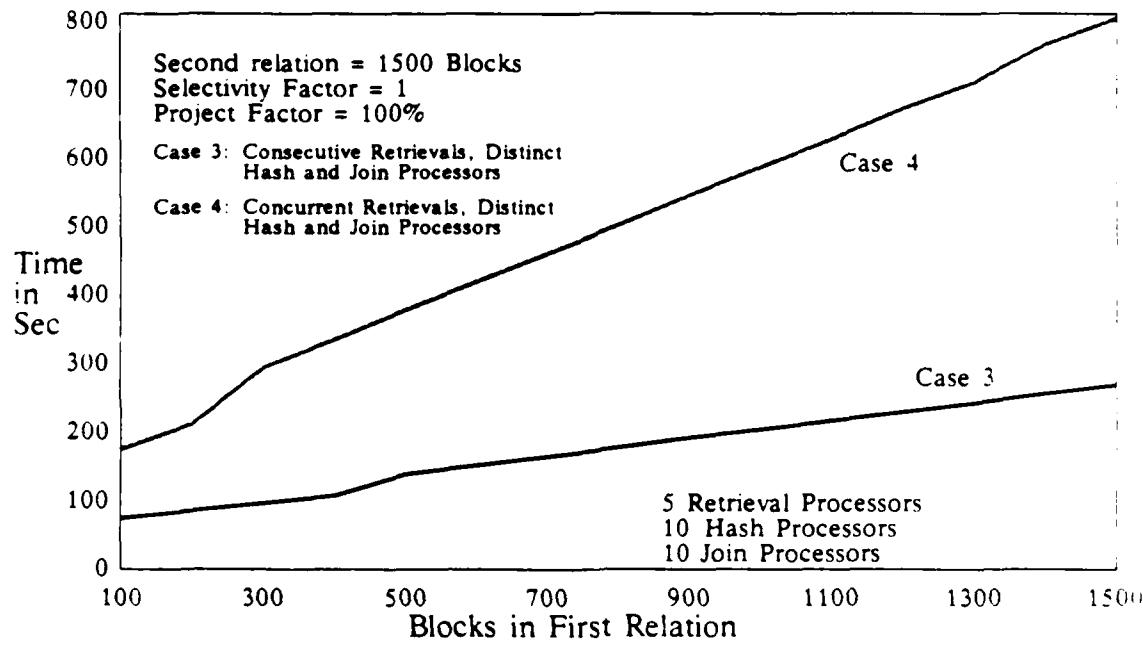


Figure 65. Multi-Step Performance using Separate Hash and Join - 3

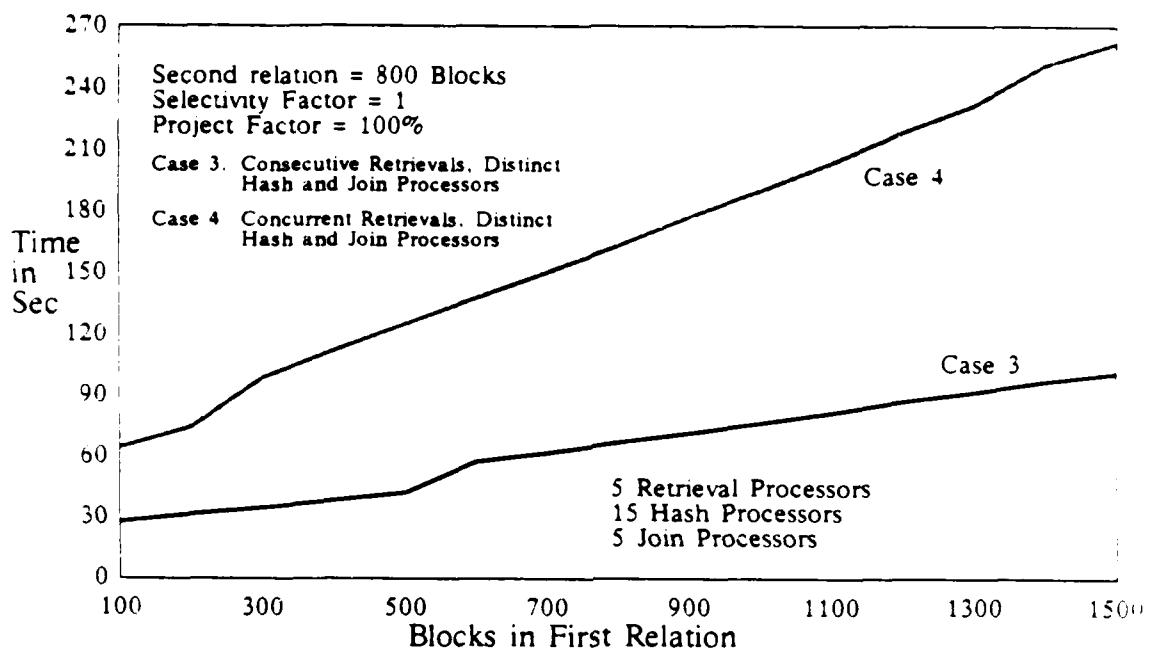


Figure 66. Multi-Step Performance using Separate Hash and Join - 4

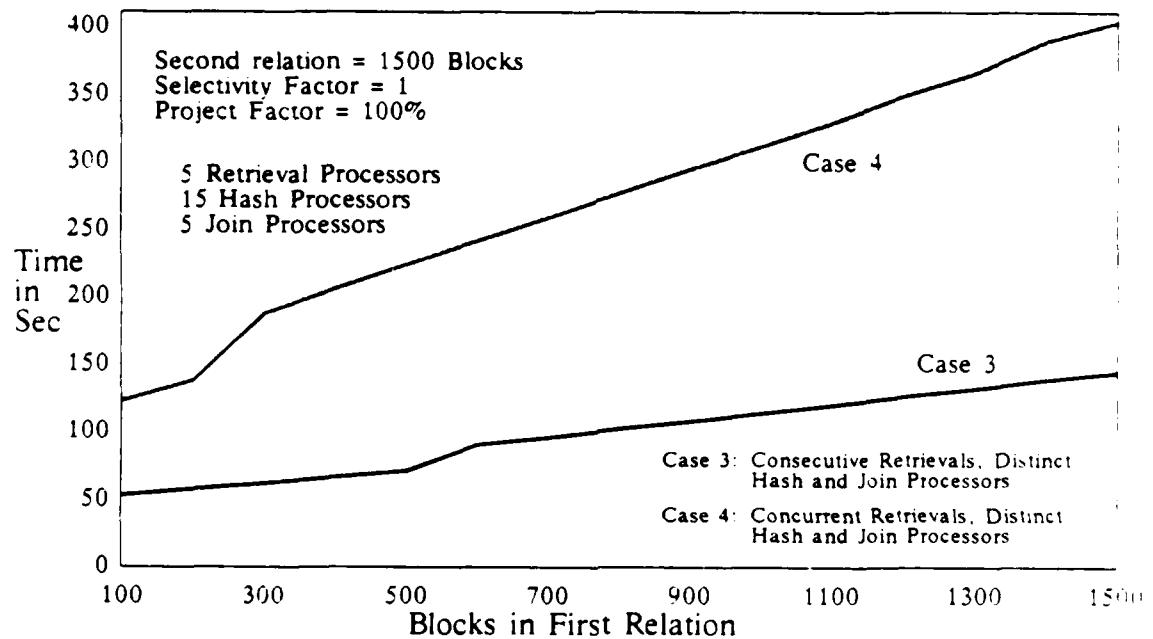


Figure 67. Multi-Step Performance using Separate Hash and Join - 5

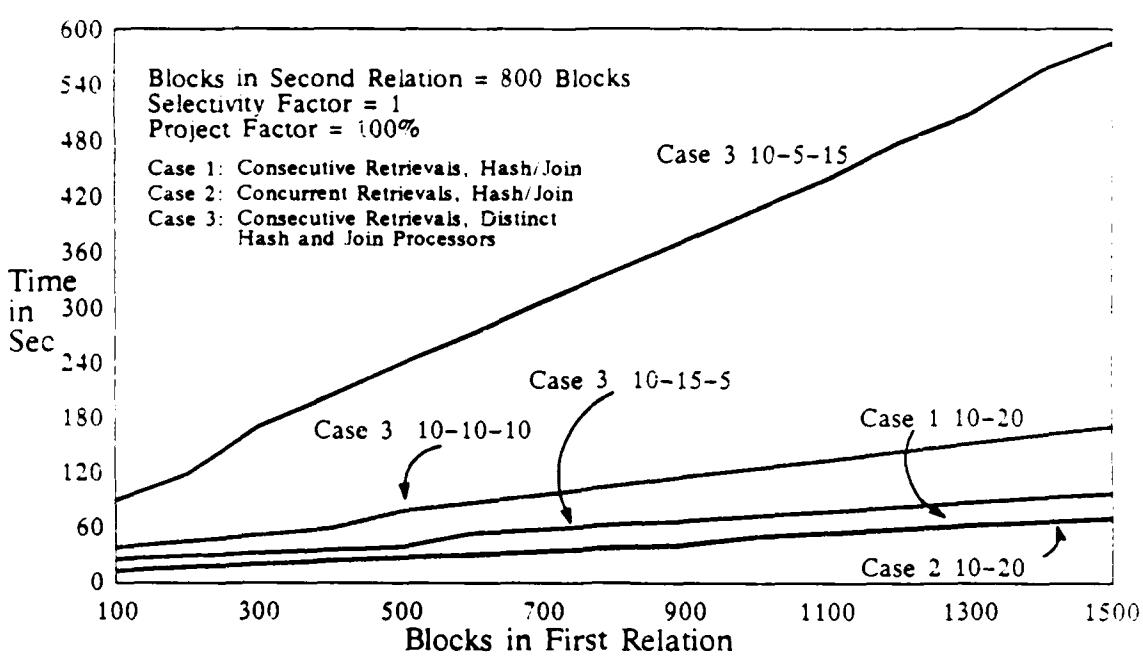


Figure 68. Performance with Selectivity Factor = 1, Projectivity Factor = 100% using 10 Retrieval Processors - 1

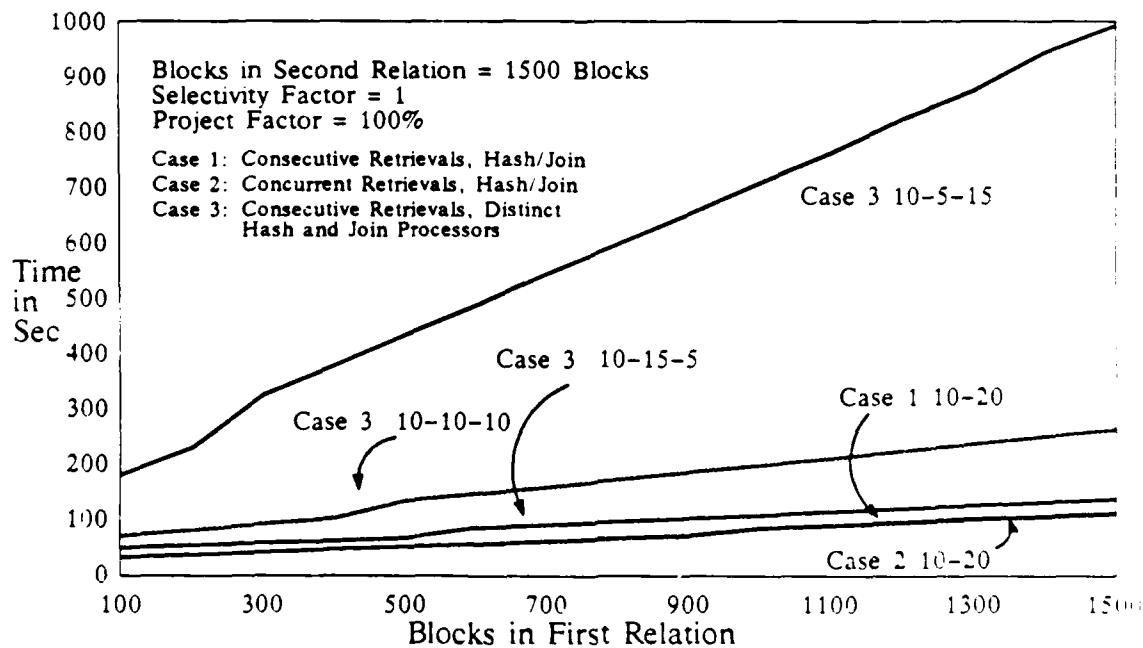


Figure 69. Performance with Selectivity Factor = 1, Projectivity Factor = 100% using 10 Retrieval Processors - 2

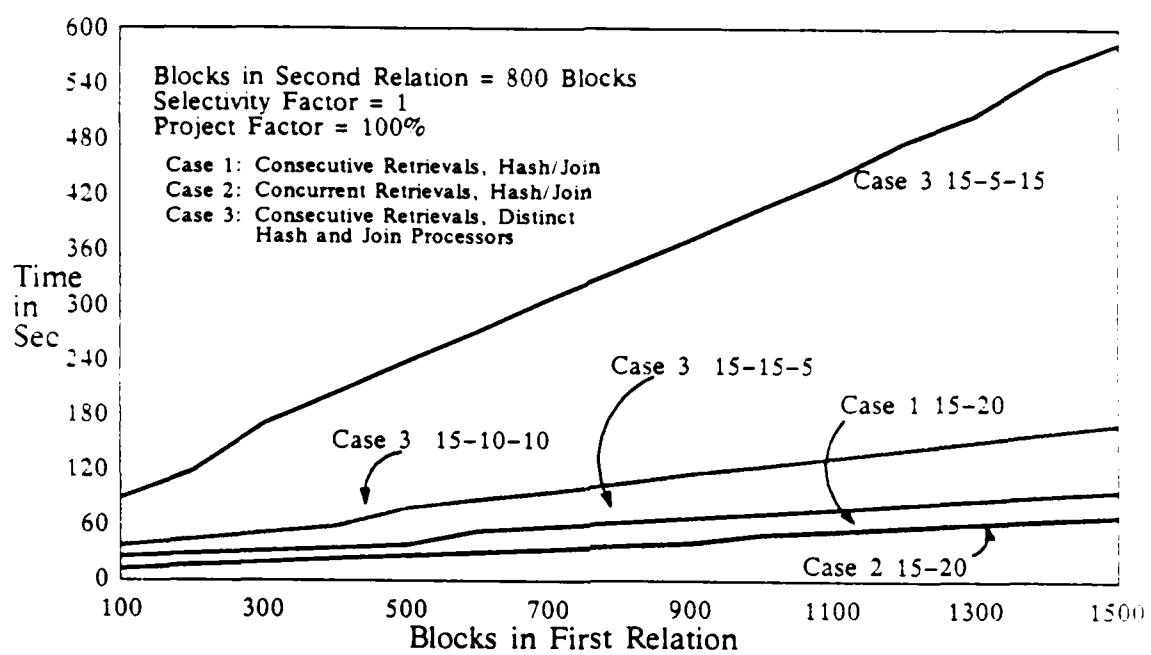


Figure 70. Performance with Selectivity Factor = 1. Projectivity Factor = 100% using 15 Retrieval Processors - 1

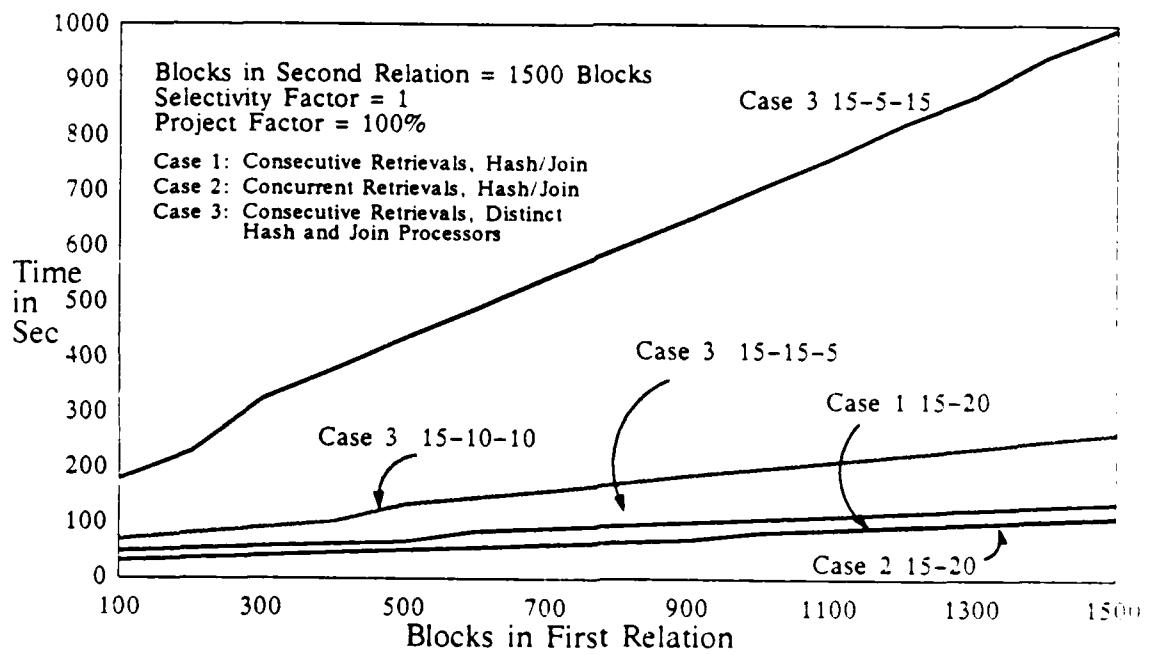


Figure 71. Performance with Selectivity Factor = 1. Projectivity Factor = 100% using 15 Retrieval Processors - 2

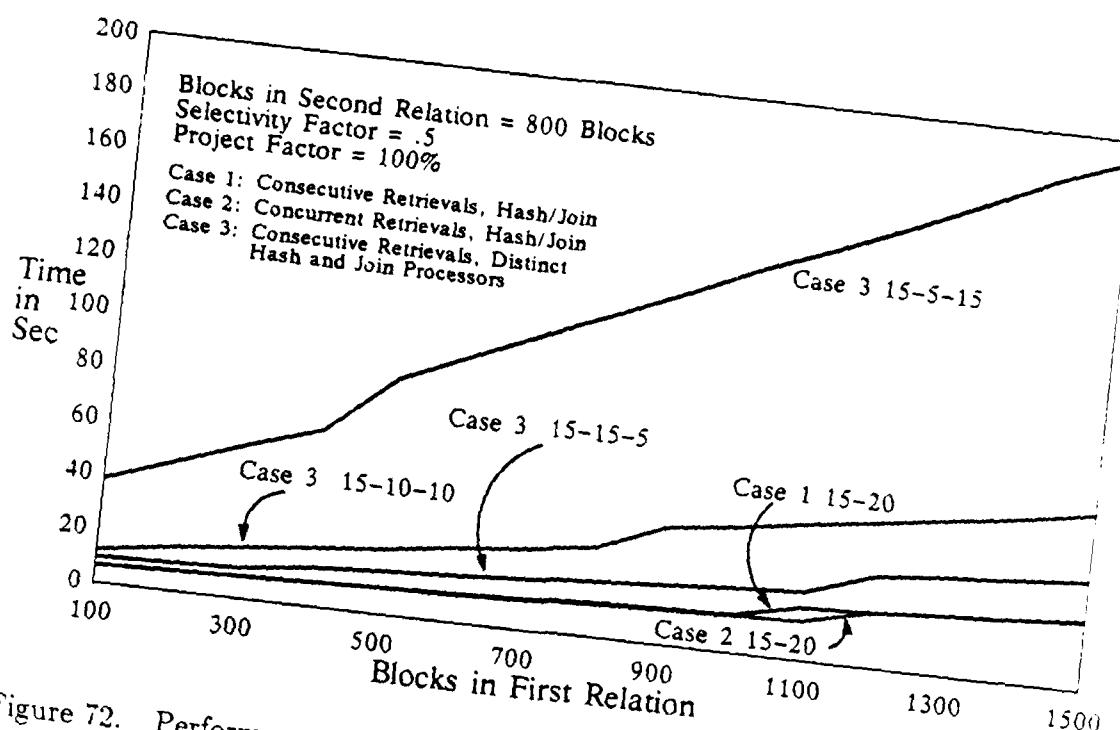


Figure 72. Performance with Selectivity Factor = .5, Projectivity Factor = 100% using 15 Retrieval Processors - 1

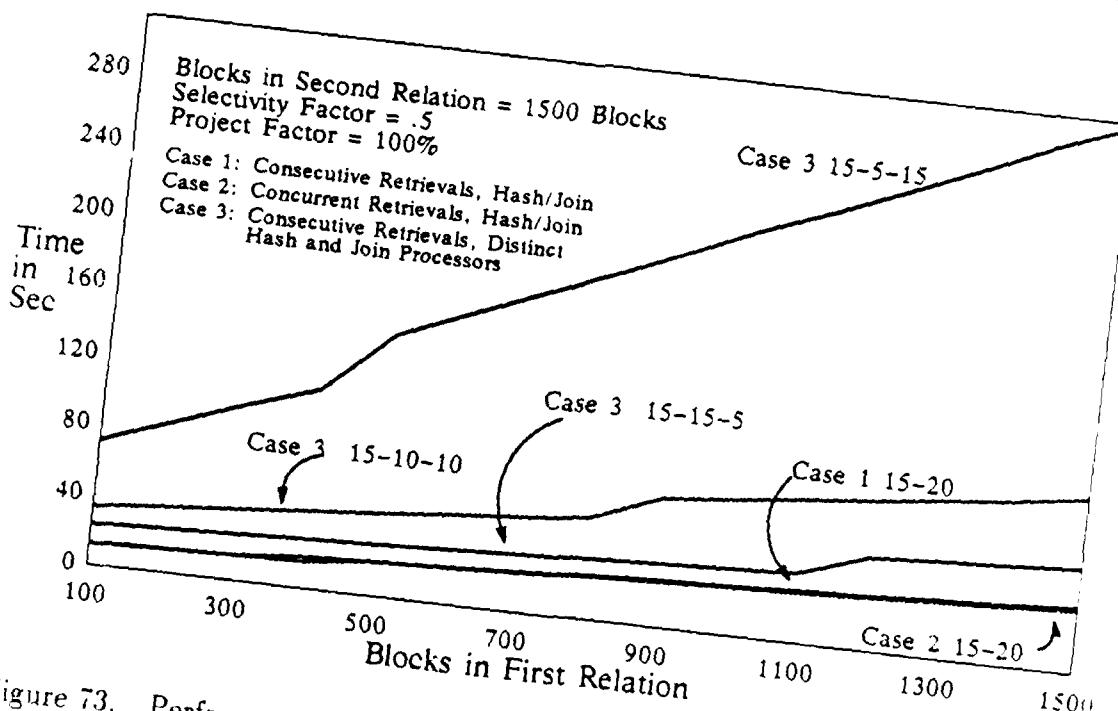


Figure 73. Performance with Selectivity Factor = .5, Projectivity Factor = 100% using 15 Retrieval Processors - 2

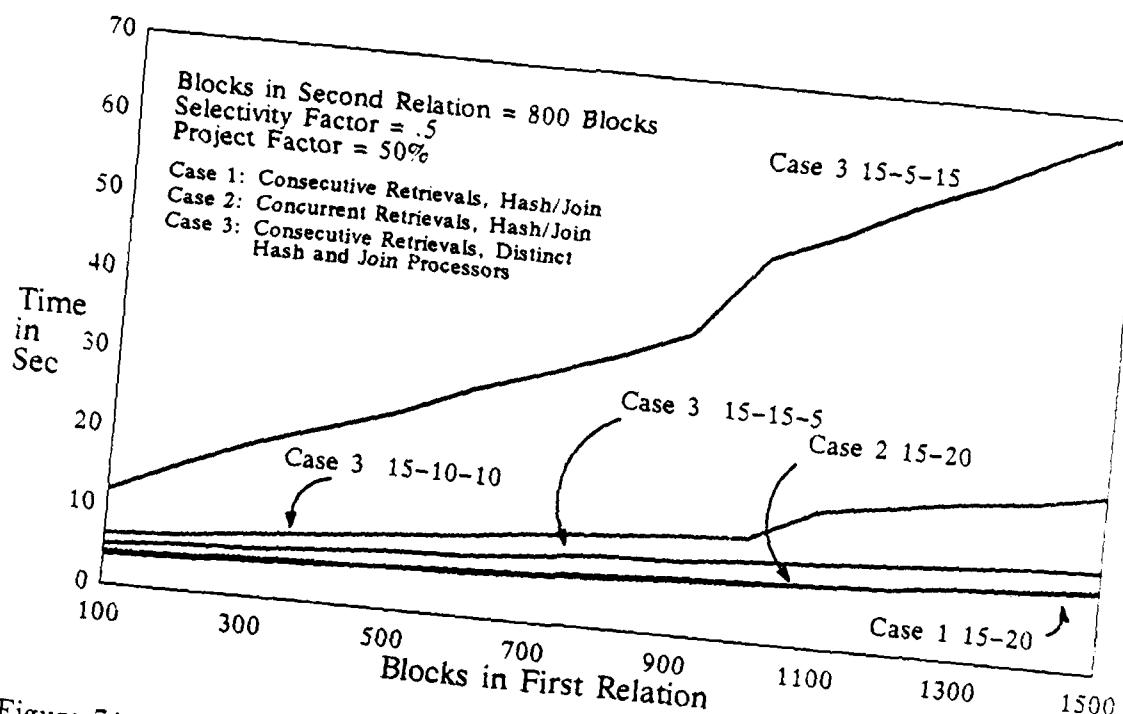


Figure 74. Performance with Selectivity Factor = .5, Projectivity Factor = 50% using 15 Retrieval Processors - 1

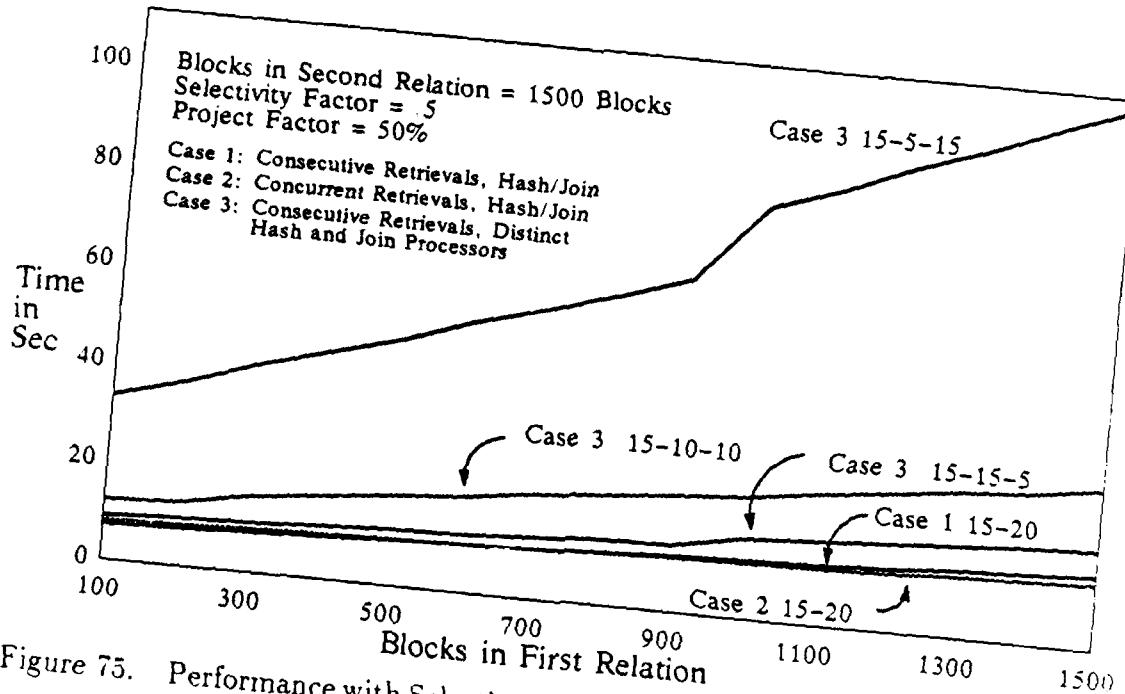


Figure 75. Performance with Selectivity Factor = .5, Projectivity Factor = 50% using 15 Retrieval Processors - 2

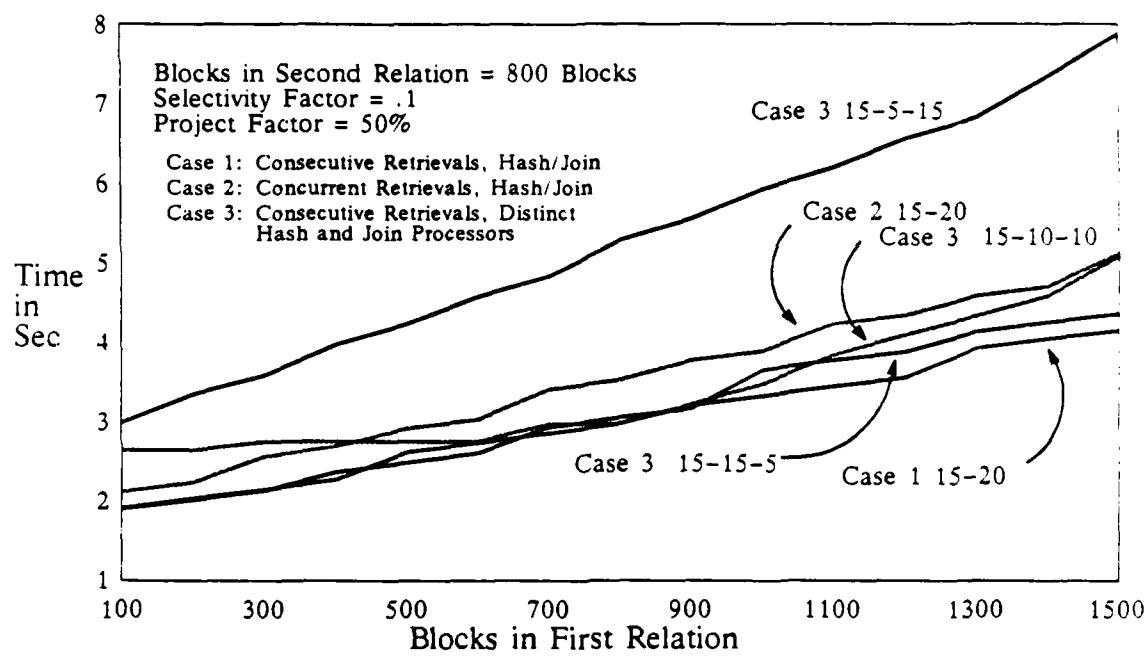


Figure 76. Performance with Selectivity Factor = .1, Projectivity Factor = 50% using 15 Retrieval Processors - 1

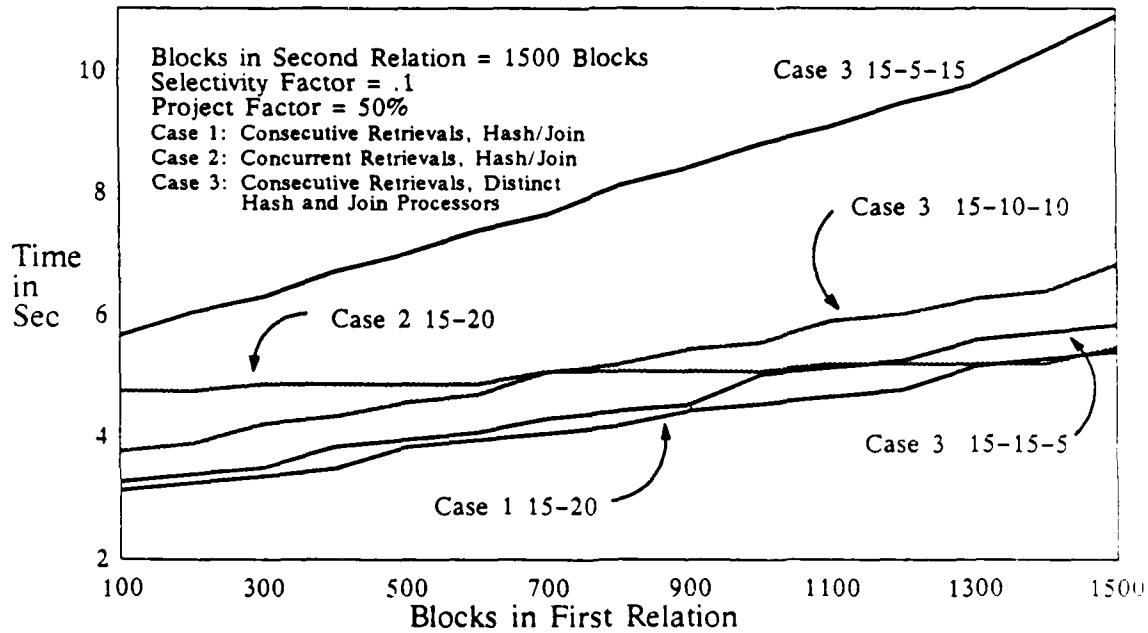


Figure 77. Performance with Selectivity Factor = .1, Projectivity Factor = 50% using 15 Retrieval Processors - 2

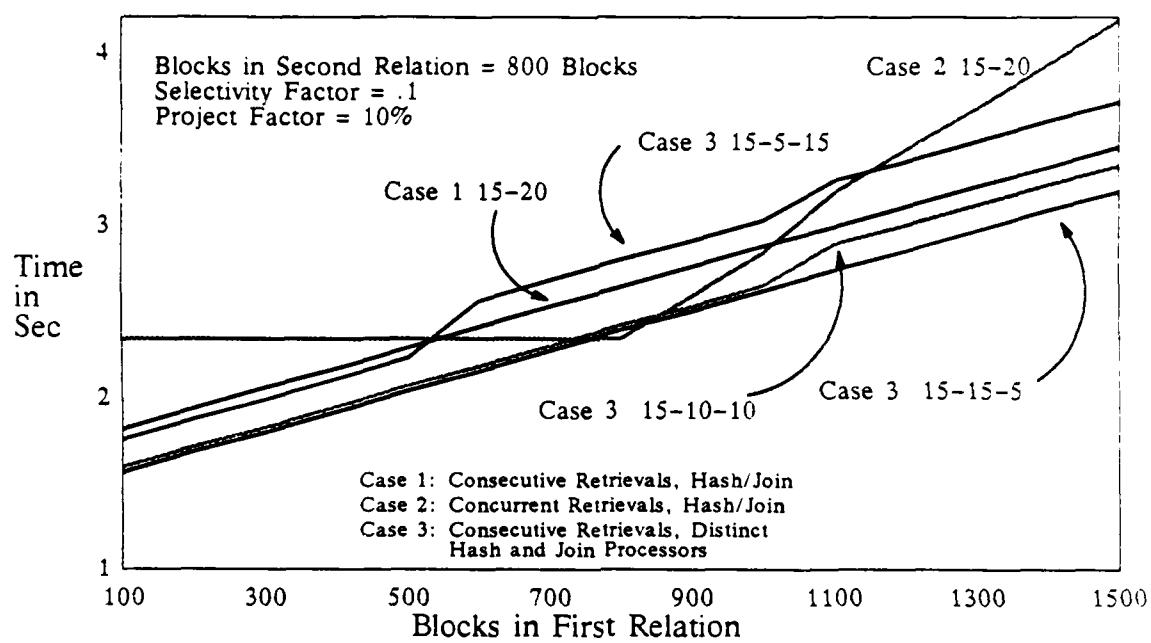


Figure 78. Performance with Selectivity Factor = .1, Projectivity Factor = 10% using 15 Retrieval Processors - 1

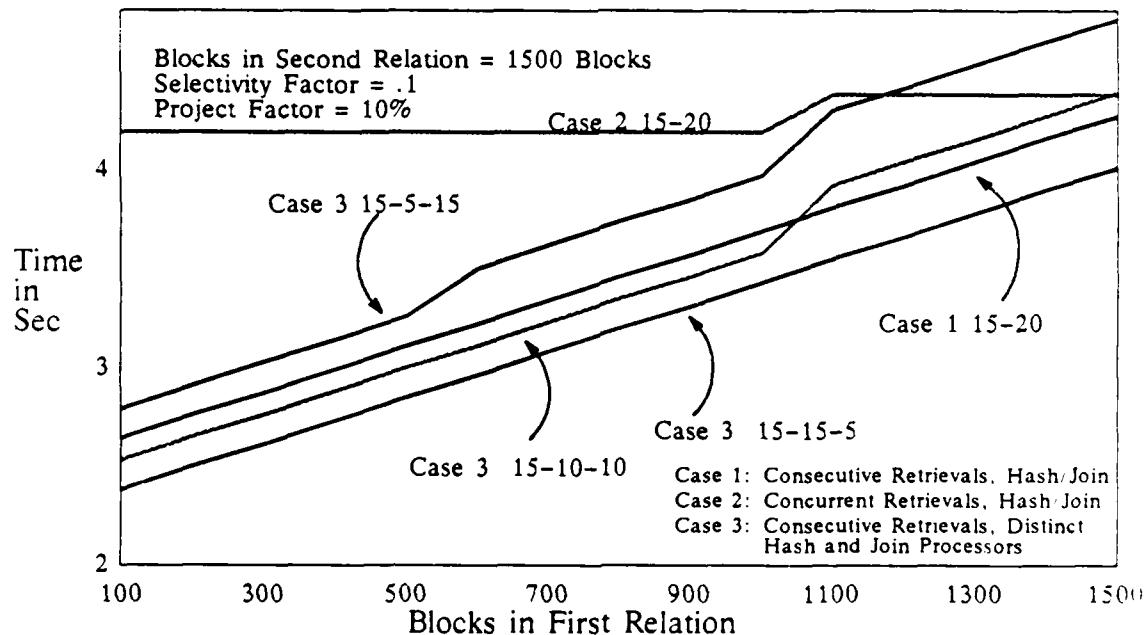


Figure 79. Performance with Selectivity Factor = .1, Projectivity Factor = 10% using 15 Retrieval Processors - 2

## Appendix B. *Multi-Binary Node Performance Results*

The multi-binary query are queries consisting of retrieving multiple relations, performing a sel-proj on the base relations, joining the results from the sel-proj, and then joining the results of the lower level joins. Figure 80 illustrates the structure of the query performance results presented here.

The multi-binary query has many parameters that could be varied in the presentation of performance results. The performance parameters that affect the size of the inputs and outputs and the number of processors and their tasks are the focus of the results presented. There are four cases presented in the following graphs. Case 1 assumes that each relation is distributed across all of the disks and that the processor assigned to process the binary nodes, use a hash/join. The hash/join assigns all of the processors to hash the incoming inputs, then when all of the input is hashed the processors begin the join of the buckets assigned to that processor. Therefore, Case 1 is annotated as Case 1 x-y on the graphs, where x is the total number of retrieval processors and y is the number of join/hash processors assigned to execute each binary node. For the Figure 80, using Case 1 10-20 means that there are 10 retrieval processors and 20 processors available for each binary node, for a total of 70 processors executing the query.

Case 2 assumes that each base relation is placed on 1/n of the disks, where n is the number of input relations. Therefore, for the model presented each relation is assumed to be distributed on 1/4 of the disks. For Case 2 10-20, this means that there are a total of 10 retrieval disks/processors are available but each relation is stored on 1/4 of the 10 disks or only 2 of the disks. The possible advantage of using the smaller number of disks to store the relations is that the retrieval of all of the relations can be overlapped. Therefore, for the model query presented both subtrees of the root node can begin processing immediately. If the data distribution of Case 1 is assumed, then the left subtree has to complete retrieving both relations before

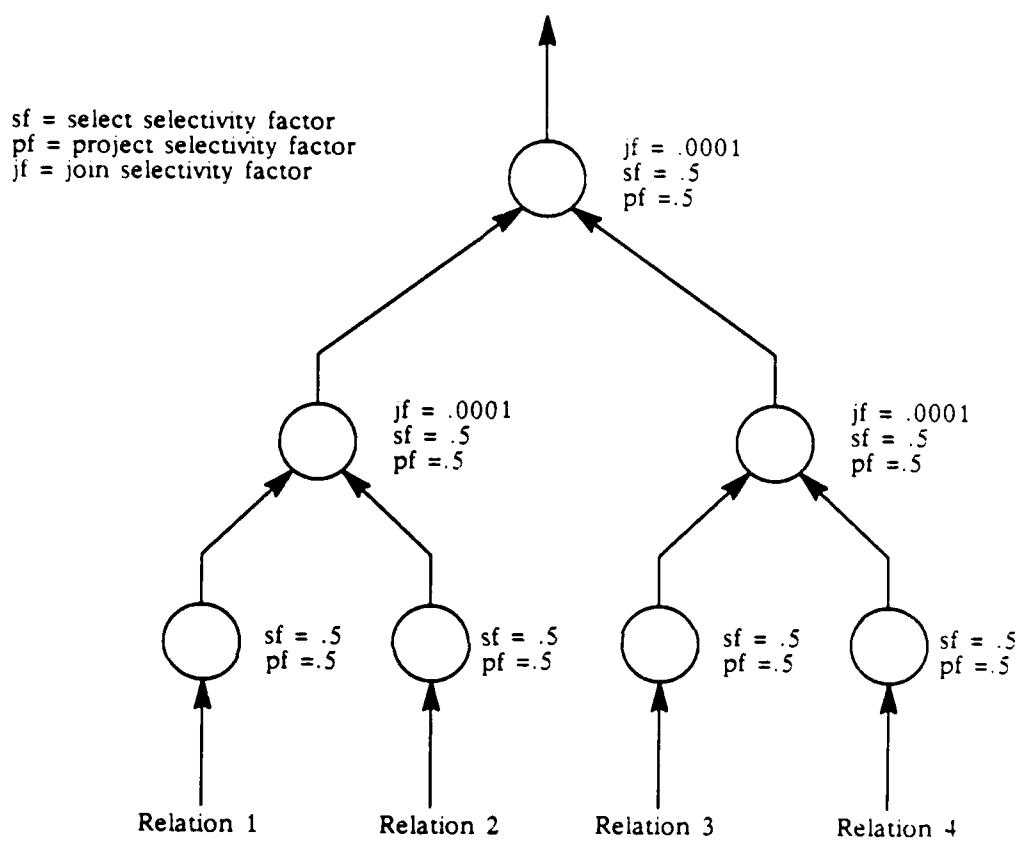


Figure 80. Multi-Binary Node Query Model

the right subtree can begin any processing. Figures 52 and 53 show this overlapping of processing possible in a multi-binary node query.

Case 2 10-20 continues by using 20 processors to hash/join at each binary node. The processor tasks are assigned the same has Case 1, where all of the processors first hash the inputs and then all of the processors join there buckets. Case 3 assumes the distribution of relations across all of the disk the same as Case 1. But, Case 3 assumes that the processors assigned to each binary node are separated so some processors only hash the inputs and the processors assigned to perform joins do not assist in the hashing of the inputs. The notation for Case 3 is Case x-y-z, where x is the number of retrieval processors, y is the number of processors hashing inputs, and z is the number of processors performing joins.

Case 4 combines the task allocation of Case 3 and the data distribution of Case 2. Therefore, Case 4 is annotated as Case 4 x-y-z on the graph. For Case 4, x is the total number of retrieval processors but each relation is stored on 1/4 of the total disks.

The results presented use the selectivity factors shown in Figure 80 to determine the size of the relations resulting from each operation. The results are presented where relation 4 size varies from 100 blocks to 1450 blocks for each graph. The other relations have their size fixed for each graph. The graphs also vary the number of processors assigned for each task for Cases 3 and 4. Only a small representative set of results are presented to show the effect of different processor allocation and the different data distributions.

The results presented illustrate the variance of the preformance depending upon the processors allocation stategy. The results do not indicate that one data allocation scheme is better than other but that the matching the number of processors to the tasks is more important. Although, the join processing is slower than hashing inputs the balance which makes it seem that more processors should be assigned to join, the results reflect the opposite of this. But the balance of using more hash pro-

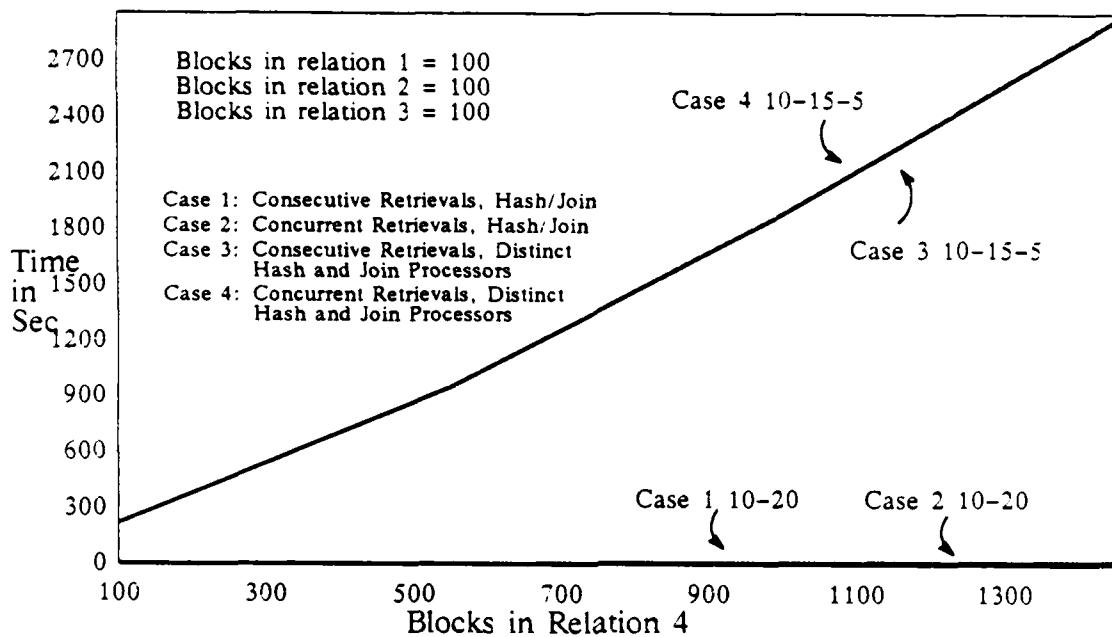


Figure 81. Multi-Binary Node Performance Graph - 1

cessors provided a better balance of processing. However, the allocation here was the same for each binary node. Futher investigation should be done to determine what the optimal number of processors at each node would provide the best performance. This should include considering varying number of processors to each binary node to try to achieve balance.

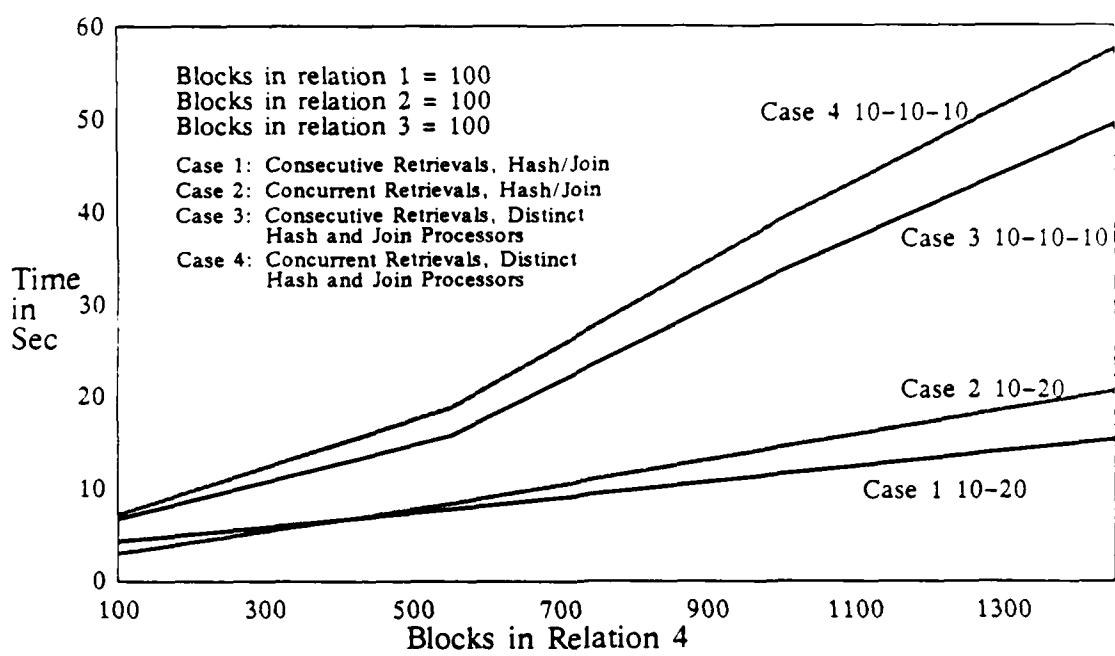


Figure 82. Multi-Binary Node Performance Graph - 2

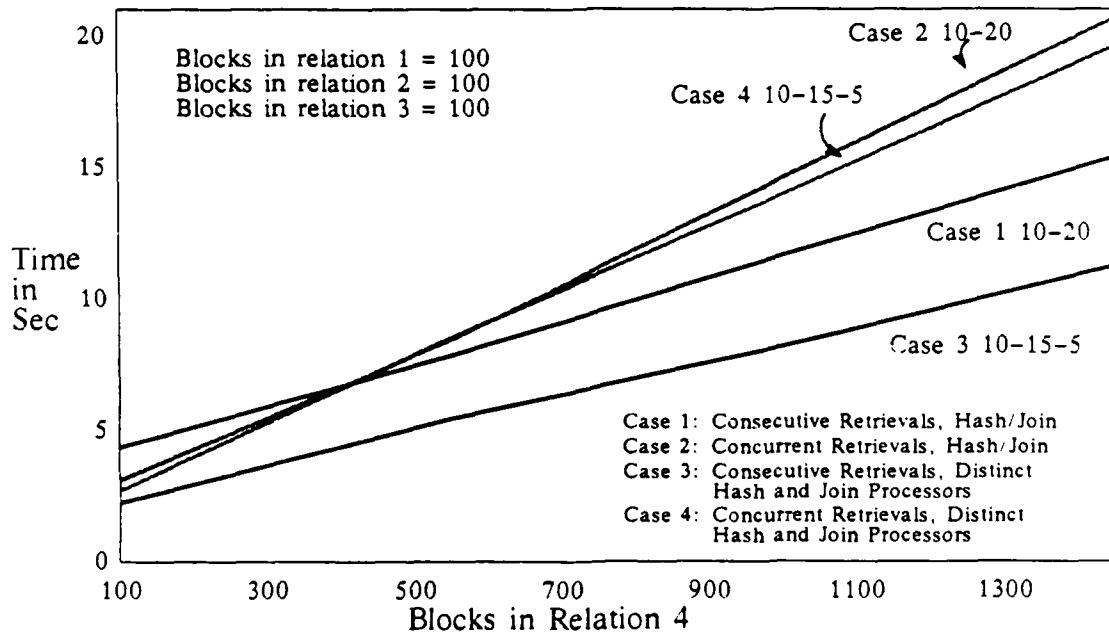


Figure 83. Multi-Binary Node Performance Graph - 3

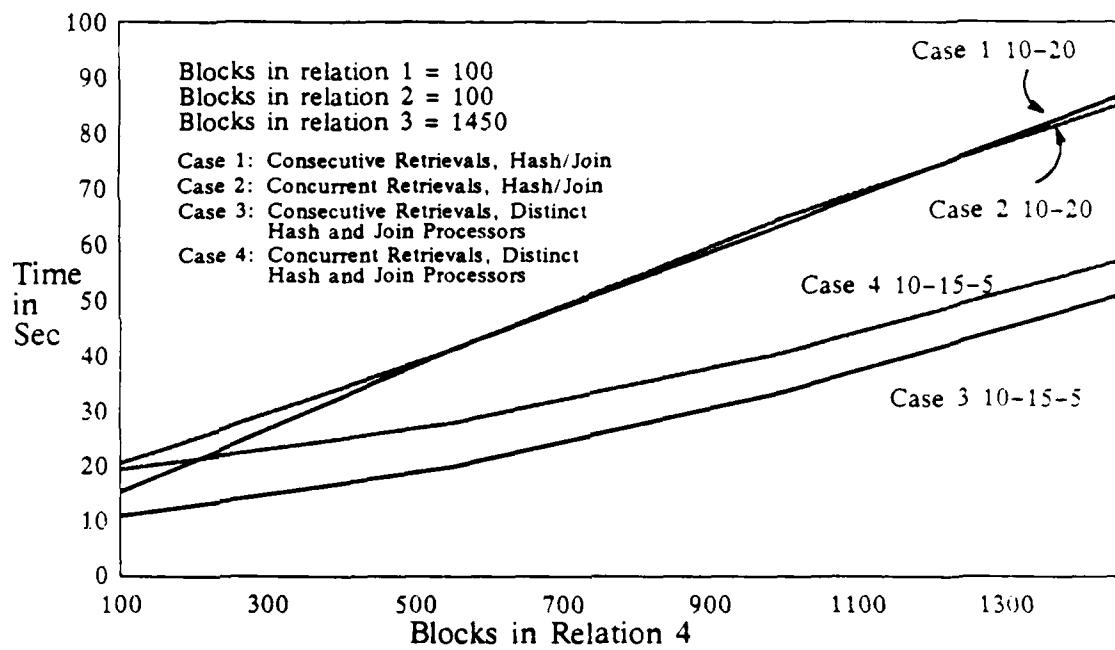


Figure 84. Multi-Binary Node Performance Graph - 4

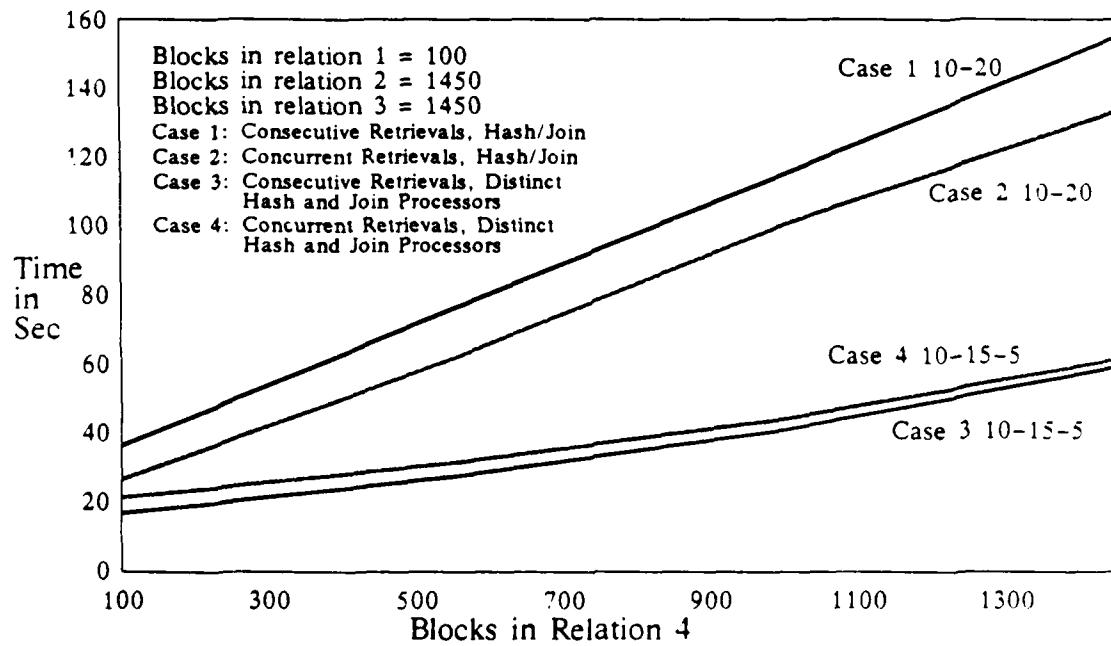


Figure 85. Multi-Binary Node Performance Graph - 5

## *Appendix C. Benchmark Performance Comparison*

Bitton, DeWitt, and Turbyfill have done a benchmarking of current database management systems and database machines [9]. This benchmark provides the possibility to compare the projected performance times for complex queries generated by the multi-step query models with actual execution times of systems. The purpose of this is not to directly compare the performance times, but to show the potential of using parallel processing in the construction of a database machine.

To compare the benchmark results with the results of the analytical models for multi-step query, some parameter translation must be done. The benchmark exercise used relations consisting of 10,000 tuples. Each tuple consisted of 182 bytes. This (using the performance parameters of Table 3) translates to a relation with approximately 140 blocks. Where each block contains 71 tuples of 182 bytes.

The next value to be considered was the number of processors to be used in computing the performance results. The benchmark compared single processor systems with multiple processor systems. However, the multiple processor system was limited to 3 or 4 processors because of controller constraints. Therefore, there was no direct comparison for the number of processors to use. The number of processors used was just an arbitrary number, but the total number of processors involved was limited to what is currently available in a multiprocessor environment. Thus, the number of processors used was 10 retrieval processors and 10 processors at each node of the complex query. This is more processors than used by any of the systems benchmarked, but the purpose of this exercise is to prove the capability of parallel processing.

Three queries were used for the benchmark. The first query, joinAselB, first select 1000 tuples from relation B and then joins this with relation A, which has 10000 tuples. The second relation assumes that the selection process on relation B

has been done prior to performing the join. Therefore, the join is a simple join of a 10,000 tuple relation and a 1000 tuple relation. The final query performs a select operation on both relations A and B that reduces the 1000 tuples each. These two 1000 tuple relations are joined and produce 1000 tuples. These 1000 tuples, from the joining of the relations A and B, are then joined with a 1000 tuple relation, C.

To approximate the number of tuples produced, a selectivity factor of .1 was used for the selections performed on A and B and a projection factor of 1 was used to provide the sel-proj operation the necessary parameters. Experience of executing the multi-step queries indicated that using a join selectivity factor of .0001 produces slightly more output blocks than the sum of the input blocks for a join operation. This is more than the number of block produced in the benchmark, but provides a value that is not less than the value used in the benchmark.

Figure 86 shows the results of the benchmark performance for the three queries. In Figure 86, Table 5 assumes the systems do not have any existing indices for the relations used. Table 6 shows the optimized retrievals for each system for performing the three queries. The performance models of the multi-step queries do not consider the data structure. Therefore, only one time parameter is expressed for each query.

The results produced for joinAselB was .08 minutes. This retrieval used 10 retrieval processors and 10 hash/join processors for a total of 20 processors. The result of joinABprime was .079 minutes. This also used a total of 20 processors. The final query joinCselAselB produced results of .083 minutes. This query utilized 10 retrieval processors and 10 processors at each binary node for a total of 40 processors. These results compare very favorable with the results presented from the benchmark in Figure 86. The results also illustrate the capability to improve the performance through the effective use of multiple processors such as the more complex query joinCselAselB being completed in approximately the same time as the less complex queries.

The results presented for the analytic performance models are not exact num-

**Join Queries Without Indices**  
**Integer Attributes**  
**Total Elapsed Time in Minutes**

System	Query		
	joinAselB	joinABprime	joinCselAselB
U-INGRES	10.2	9.6	9.4
C-INGRES	1.8	2.6	2.1
ORACLE	>300	>300	>300
IDMnodac	>300	>300	>300
IDMdac	>300	>300	>300
DIRECT	10.2	9.5	5.6

**Join Queries With Indices**  
**Integer Attributes**  
**Total Elapsed Time in Minutes**  
**Primary (clustered) Index on Join Attribute**

System	Query		
	joinAselB	joinABprime	joinCselAselB
U-INGRES	2.11	1.66	9.07
C-INGRES	0.90	1.71	1.07
ORACLE	7.94	7.22	13.78
IDMnodac	0.52	0.59	0.74
IDMdac	0.39	0.46	0.58
DIRECT	10.21	9.47	5.62

**Join Queries With Indices**  
**Total Elapsed Time in Minutes**  
**Secondary (nonclustered) Index on Join Attribute**

System	Query		
	joinAselB	joinABprime	joinCselAselB
U-INGRES	4.49	3.24	10.55
C-INGRES	1.97	1.80	2.41
ORACLE	8.52	9.39	18.85
IDMnodac	1.41	0.81	1.81
IDMdac	1.19	0.59	1.47
DIRECT	10.21	9.47	5.62

**Figure 86. Benchmark Results**  
**(Reproduced from Reference [9])**

bers but an estimation based upon a given set of performance parameters. The performance parameters used are easily obtainable values for today's state-of-the-art hardware. The only definable difference between the benchmark cases and the performance models is the use of multiple processors utilizing parallel processing. This comparison encourages the modeling and design considerations presented in this document to be extended to a physical implementation to provide the capability for further analysis of the effectiveness of parallelism in database operations.

## Bibliography

1. ——. The Genesis of a Database Computer. *Computer*, 17(11):42–56, November 1984.
2. E. Babb. Implementing a Relational Database by Means of Specialized Hardware. *ACM Transactions on Database Systems*, 4(1):1–29, March 1979.
3. F. Bancilhon et al. VERSO: A Relational Backend Database Machine. In David K. Hsiao, editor, *Advanced Database Architecture*. Prentice-Hall, 1983.
4. Janyanta Banerjee, David J. Hsiao, and Krishnamurthi Kannar. DBC - A Database Computer for Very Large Databases. *IEEE Transactions on Computers*, C-28(6):414–429, June 1979.
5. Chaitanya K. Baru and Stanley Y. W. Su. The Architecture of SM3: A Dynamically Partitionable Multicomputer System. *IEEE Transactions on Computers*, C-35(9):790–802, September 1986.
6. P. Bruce Berra and Ellen Oliver. The Role of Associative Array Processors in Data Base Machine Architecture. *Computer*, 12(3):53–61, March 1979.
7. Dina Bitton, Haran Boral, David DeWitt, and W. Kevin Wilkinson. Parallel Algorithms for the Execution of Relational Database Operations. *ACM Transactions on Database Systems*, 8(3):324–353, September 1983.
8. Dina Bitton and David J. DeWitt. Duplicate Record Elimination in Large Data Files. *ACM Transactions on Database Systems*, 8(2):255–265, June 1983.
9. Dina Bitton, David J. DeWitt, and Carolyn Turbyfill. *Benchmarking Database Systems, A Systematic Approach*. Computer Science Technical Report 526. University of Wisconsin - Madison, 1983.
10. Taylor Booth, et al. Design Education in Computer Science and Engineering. *Computer*, 19(6):20–27, June 1986.
11. Haran Boral. Database Machine Morphology. In *Proceedings of Eleventh International Conference of Very Large Databases*. A. Pirotte and Y. Vassiliou, editors, 1985.
12. Haran Boral and David J. DeWitt. Applying Data Flow Techniques to Data Base Machines. *Computer*, 15(8):57–63, August 1982.
13. Haran Boral and David J. DeWitt. Database Machines: An Idea Whose Time has Passed? A Critique of the Future of Database Machines. In *Database Machines*, Springer-Verlag, H.-O. Leilich and M. Missikoff, editors, 1983.
14. Haran Boral and David J. DeWitt. *Design Considerations for Data-Flow Database Machines*. MRC Technical Summary Report 2058. University of Wisconsin-Madison, 1980.

15. Haran Boral and David J. DeWitt. Processor Allocation Strategies for Multiprocessor Database Machines. *ACM Transactions on Database Systems*. 6(2):227-254, June 1981.
16. F. Cesarini, F. Pippolini, and G. Soda. A Technique for Analyzing Query Execution in a Multiprocessor Database Machine. In *Database Machines, Proceedings of Fourth International Workshop on Database Machines*, D. J. DeWitt and H. Boral, editors, pages 68-90, Springer-Verlag, 1985.
17. E.F. Codd. A Relation Extending the Database Relational Model to Capture More Meaning. *ACM Transactions on Database Systems*, 4:397-434, December 1979.
18. E.F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13:377-387, June 1970.
19. C. J. Date. *An Introduction to Database Systems*, 3rd ed. Addison-Wesley, 1981.
20. C. J. Date. *An Introduction to Database Systems, Volume II*. Addison-Wesley, 1983.
21. David J. DeWitt. DIRECT - A Multiprocessor Organization for Supporting Relational Database Management Systems. *IEEE Transactions on Computers*. C-28(6):395-406, June 1979.
22. David J. DeWitt. GAMMA - A High Performance Dataflow Database Machine. In *Proceedings of the Twelfth International Conference on Very Large Databases*. Morgan-Kaufmann Publishers, Inc., 1986.
23. David J. DeWitt. Query Execution in DIRECT. In *Proceedings of 1979 ACM SIGMOD International Conference on Management of Data*, pages 13-22, 1979.
24. David J. DeWitt and Robert Gerber. Multiprocessor Hash-Based Join Algorithms. In *Proceedings of Eleventh International Conference on Very Large Databases*, 1985.
25. David J. DeWitt and Paula B. Hawthorn. A Performance Evaluation of Database Machine Architectures. In *Proceedings of 7th International Conference on Very Large Data Bases*, pages 199-215, IEEE Computer Society Press, 1981.
26. David J. DeWitt, Randy H. Katz, et al. Implementation Techniques for Main Memory Database Systems. *SIGMOD Record*. 14(2):1-8, June 1984.
27. H. C. Du. Distributing a Database for Parallel Processing is NP-Hard. *ACM SIGMOD Record*. 14(1):55-60, March 1984.
28. Robert Epstein and Paula Hawthorn. Aid in the '80s. *Datamation*, 154-158, February 1980.

29. Robert Epstein and Paula Hawthorn. Design Decisions for the Intelligent Database Machine. In *AFIPS Conference Proceedings, Volume 49*, pages 237–241. AFIPS Press, 1984.
30. Daniel H. Fishman, Ming-Yee Lai, and W. Kevin Wilkinson. Overview of the Jasmin Database Machine. *ACM SIGMOD Record*, 14(2):234–239, June 1984.
31. Robert W. Fonden. *Design and Implementation of a Backend Multiple-Processor Relational Data Base Computer System*. Master's thesis, Air Force Institute of Technology (AU), Wright-Patterson AFB, Ohio, 1981. (AD-A115558).
32. Georges Gardarin, et al. SABRE: A Relational Database System for a Multi-microprocessor Machine. In *Advanced Database Architecture*, David K. Hsiao, editor, Prentice-Hall, 1983.
33. Paula B. Hawthorn and David J. DeWitt. Performance Analysis of Alternative Database Machine Architectures. *IEEE Transactions on Software Engineering*. SE-8(1):61–75, January 1982.
34. Xin-Gui He, et al. The Implementation of a Multibackend Database System (MDBS): Part II - The Design of a Prototype MDBS. In *Advanced Database Architecture*, David K. Hsiao, editor, Prentice-Hall, 1983.
35. Michael Heagney, Lee Wieland and Jane Betz, and Larry Gosselin. *Omnibase Test Results*. Technical Report, Marathon Oil Company, 1983.
36. Bruce K. Hillyer, David Elliot Shaw, and Anil Nigam. NON-VON's Performance on Certain Database Benchmarks. *IEEE Transactions on Software Engineering*. SE-12(4):577–583, April 1986.
37. Yang-Chang Hong. Efficient Computing of Relational Algebraic Primitives in a Database Machine Architecture. *IEEE Transactions on Computers*. C-34(7):588–595, July 1985.
38. Yang-Chang Hong. A Pipeline and Parallel Architecture for Supporting Database Management Systems. In *Proceedings of 1984 International Conference on Data Engineering*, pages 152–159, IEEE Computer Society Press, 1984.
39. Ellis Horowitz and Sartaj Sahni. *Fundamentals of Data Structures*. Computer Science Press, 1982.
40. David K. Hsiao. Cost-Effective Ways of Improving Database Computer Performance. In *AFIPS Conference Proceeding 52, National Computer Conference 1983*, AFIPS Press, 1983.
41. David K. Hsiao. Data Base Machines are Coming, Data Base Machines are Coming. *Computer*, 12(3):7–10, March 1979.
42. David K. Hsiao, et al. The Implementation of a Multibackend Database System (MDBS): Part I - An Exercise in Database Software Engineering. In *Advanced Database Architecture*, David K. Hsiao, editor, Prentice-Hall, 1983.

43. David K. Hsiao and M. Jaishanker Menon. *Design and Analysis of a Multi-Backend Database System For Performance Improvement, Functionality Expansion and Capacity Growth (Part I)*. Technical Report, Computer and Information Science Research Center, Ohio State University, Columbus, Ohio. 1981.
44. Shigeo Kamiya, et al. A Hardware Pipeline Algorithm for Relational Database Operation and Its Implementation using Dedicated Hardware. In *Proceedings of 12th Annual Symposium on Computer Architecture*, pages 250–257, IEEE Computer Society Press, 1985.
45. R. M. Karp and R. E. Miller. Properties of a Model for Parallel Computations: Determinacy, Termination, Queueing. *SIAM Journal of Applied Math.* 14(6):1390–1411, November 1966.
46. Douglas S. Kerr. Data Base Machines with Large Content-Addressable Blocks and Structural Information Processors. *Computer*, 12(3):64–79, March 1979.
47. Won Kim, Daniel Gajski, and David J. Kuck. A Parallel Pipelined Relational Query Processor. *ACM Transactions on Database Systems*, 9(2):214–242, June 1984.
48. Masaru Kitsuregawa, Hidehiko Tanaka, and Tohru Moto-oka. Architecture and Performance of Relational Algebra Machine GRACE. In *Proceedings of 1984 International Conference on Parallel Processing*, pages 241–250, IEEE Computer Society Press, 1984.
49. Jr. Langdon, Glen G. A Note on Associative Processors for Data Management. *ACM Transactions on Database Systems*, 3(2):148–158, June 1978.
50. G. J. Lipovski. Architectural Features of CASSM: A Context Addressed Segment Sequential Memory. In *Proceedings of the 5th Annual Symposium on Computer Architecture*, pages 31–38, IEEE Computer Society Press, 1978.
51. Frank J. Malabarba. Data Base Machines - It's About Time! In *Proceedings Trends and Applications 1982*, pages 19–25, IEEE Computer Society Press. 1982.
52. Frank J. Malabarba. Review of Available Database Machine Technology. In *Proceedings Trends and Applications 1984*, pages 14–17, IEEE Computer Society Press, 1984.
53. M. J. Menon and David K. Hsiao. Design and Analysis of Join Operations of Database Machines. In *Advanced Database Architecture*, David K. Hsiao. editor. Prentice-Hall, 1983.
54. Timothy H. Merrett. *Relational Information Systems*. Reston Publishing, 1984.
55. Raymond E. Miller. A Comparison of Some Theoretical Models of Parallel Computation. *IEEE Transactions on Computers*, C-22(8):710–717, August 1973.

56. M. Missikoff and M. Terranova. The Architecture of a Relational Database Computer Known as DBMAC. In *Advanced Database Architecture*, David K. Hsiao, editor, Prentice-Hall, 1983.
57. Keith Morgan. The Intel Database Processor. In *Proceedings of 1983 Spring CompCon*, pages 371–373, IEEE Computer Society Press, 1983.
58. T. Nakayama, M. Hirakawa, and T. Ichikawa. Architecture and Algorithm for Parallel Execution of a Join Operation. In *Proceeding of 1984 International Conference on Data Engineering*, pages 160–165, IEEE Press, 1984.
59. Philip M. Neches. Hardware Support for Advanced Data Management Systems. *Computer*, 17(11):29–41, November 1984.
60. Esen Ozkarahan. *Database Machines and Database Management*. Prentice-Hall, 1986.
61. G. Pelgatti and F. A. Schriber. A Model of an Access Strategy in a Distributed Database. In *Database Architecture: Proceedings of IFIP-TC2*, G. Bracchi and G. M. Nijssen, editors, North-Holland Publishing, 1979.
62. Dale M. Pontiff. *Backend Control Processor for a Multi-Processor Relational Database Computer System*. Master's thesis, Air Force Institute of Technology (AU), Wright-Patterson AFB, Ohio, 1984. (AD-A151982).
63. S. Pramanik and M. H. Kim. HCB-Tree: A B-Tree Structure for Parallel Processing. In *Proceedings of 1987 International Conference on Parallel Processing*, Sartaj K. Sahni, editor, pages 140–146, The Pennsylvania State University Press, 1987.
64. G. Z. Qadah and K. B. Irani. A Database Machine for Very Large Relational Databases. In *Proceedings of 1983 International Conference on Parallel Processing*, pages 307–314, IEEE Computer Society Press, 1983.
65. C. V. Ramamoorthy. *Design Methodology for Back-End Database Machines in Distributed Environments*. Technical Report for U.S. Army Research Office, Electronics Research Laboratory, University of California, Berkeley, California, 1984. (AD-A142122).
66. William R. Rogers. *Development of a Query Processor for a Back-End Multi-processor Relational Database Computer*. Master's thesis, Air Force Institute of Technology (AU), Wright-Patterson AFB, Ohio, 1982. (AD-A124885).
67. Roger K. Schultz and Roy J. Zingg. Response Time Analysis of Multiprocessor Computers for Database Support. *ACM Transactions on Database Systems*, 9(1):100–132, March 1984.
68. S. A. Schuster, et al. RAP.2 - An Associative Processor for Data Bases. In *Proceedings of the 5th Annual Symposium on Computer Architecture*, pages 31–38, IEEE Computer Society Press, 1978.

69. S. A. Schuster, et al. RAP.2 - An Associative Processor for Databases and Its Applications. *IEEE Transactions on Computers*, C-28(6):446-458, June 1979.
70. H. Schweißpe, et al. RDBM - A Dedicated Multiprocessor System for Database Management. In *Advanced Database Architecture*, David K. Hsiao, editor. Prentice-Hall, 1983.
71. Akira Sekino, et al. Design and Implementation of an Information Query Computer. In *Proceedings of 1983 Spring CompCon*, pages 374-377, IEEE Computer Society Press, 1983.
72. Diane C. P. Smith and John Miles Smith. Relational Data Base Machines. *Computer*, 12(3):28-38, March 1979.
73. J. M. Smith and P. Yen-Tang Chang. Optimizing the Performance of a Relational Algebra Interface. *Communications of the ACM*, 18(10):568-579, October 1975.
74. David L. Spooner and Ehud Gudes. A Unifying Approach to the Design of a Secure Database Operating System. *IEEE Transactions on Software Engineering*. SE-10(3):310-319, May 1984.
75. Donald F. Stanat and David F. McAllister. *Discrete Mathematics in Computer Science*. Prentice-Hall, 1977.
76. Michael Stonebraker. Operating System Support for Database Management. *Communications of the ACM*, 24(7):412-418, July 1981.
77. Stanley Y. W. Su. Cellular-Logic Devices: Concepts and Applications. *Computer*, 12(3):11-25, March 1979.
78. J. Tuazon, et al. Caltech/JPL Mark II Hypercube Concurrent Processor. In *Proceedings of 1984 International Conference on Parallel Processing*, pages 666-673, IEEE Computer Society Press, 1984.
79. Jeffrey D. Ullman. *Principles of Database Systems*, 2nd Ed. Computer Science Press, 1982.
80. Patrick Valduriez and Georges Gardarin. Join and Semijoin Algorithms for a Multiprocessor Database Machine. *ACM Transactions on Database Systems*. 9(1):133-161, March 1984.
81. He Xinqui. A Process-Based Data Flow Database Machine (PDDM). In *Proceedings of The First International Conference on Computers and Applications*, pages 608-615, IEEE Computer Society Press, 1984.
82. S. B. Yao. Optimization of Query Evaluation Algorithms. *ACM Transactions on Database Systems*, 4(2):133-155, June 1979.
83. S. Bing Yao and Alan R. Hevner. *Benchmark Analysis of Database Architectures: A Case Study*. NBS Special Publication 500-132, U.S. Dept of Commerce, National Bureau of Standards, 1984.

84. S. Bing Yao, Alan R. Hevner, and Shao T. Yu. A Performance Benchmark of a Database Machine. In *Proceedings Trends and Applications 1984*, pages 125–131, IEEE Computer Society Press, 1984.

### *Vita*

Timothy G. Kearns was born on 11 September 1952 in Rushville, Nebraska. He graduated from high school in Rushville, Nebraska in 1970 and attended Chadron State College, Chadron, Nebraska. He received a Bachelor of Science in Math Education in 1974. He completed OTS in September of 1979, receiving a commission in the United States Air Force. From October of 1979 through May 1983 he worked at HQ AFLC, Wright-Patterson AFB, Ohio, as a computer programmer and software engineer. He entered the School of Engineering, Air Force Institute of Technology, in May 1983 and was awarded a Master of Science in Computer Systems degree in December 1984.

Permanent address: HC 65 Box 77  
Rushville, Nebraska 69360

UNCLASSIFIED  
SECURITY CLASSIFICATION OF THIS PAGE

Form Approved  
OMB No. 0704-0188

REPORT DOCUMENTATION PAGE

1. REPORT SECURITY CLASSIFICATION <b>UNCLASSIFIED</b>		1b. RESTRICTIVE MARKINGS										
2a. SECURITY CLASSIFICATION AUTHORITY		3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited.										
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE												
4. PERFORMING ORGANIZATION REPORT NUMBER(S)  AFIT/DS/ENG/87-1		5. MONITORING ORGANIZATION REPORT NUMBER(S)										
6a. NAME OF PERFORMING ORGANIZATION  School of Engineering	6b. OFFICE SYMBOL <i>(If applicable)</i>  AFIT/ENG	7a. NAME OF MONITORING ORGANIZATION										
6c. ADDRESS (City, State, and ZIP Code)  Air Force Institute of Technology Wright-Patterson AFB OH 45433-6583		7b. ADDRESS (City, State, and ZIP Code)										
8a. NAME OF FUNDING/SPONSORING ORGANIZATION	8b. OFFICE SYMBOL <i>(If applicable)</i>	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER										
8c. ADDRESS (City, State, and ZIP Code)		10. SOURCE OF FUNDING NUMBERS <table border="1"><tr><td>PROGRAM ELEMENT NO.</td><td>PROJECT NO.</td><td>TASK NO</td><td>WORK UNIT ACCESSION NO.</td></tr></table>		PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO	WORK UNIT ACCESSION NO.					
PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO	WORK UNIT ACCESSION NO.									
11. TITLE (Include Security Classification)  See Box 19												
PERSONAL AUTHOR(S) Timothy G. Kearns, B.S., M.S., Capt, USAF												
13a. TYPE OF REPORT  Ph.D. Dissertation	13b. TIME COVERED FROM _____ TO _____	14. DATE OF REPORT (Year, Month, Day) 1987 December	15. PAGE COUNT 349									
16. SUPPLEMENTARY NOTATION												
17. COSATI CODES <table border="1"><tr><th>FIELD</th><th>GROUP</th><th>SUB-GROUP</th></tr><tr><td>12</td><td>08</td><td></td></tr><tr><td></td><td></td><td></td></tr></table>		FIELD	GROUP	SUB-GROUP	12	08					18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)  Computer, Data Bases, Data Management, Computer Architecture	
FIELD	GROUP	SUB-GROUP										
12	08											
19. ABSTRACT (Continue on reverse if necessary and identify by block number)  Title: A METHODOLOGY, BASED ON ANALYTICAL MODELING, FOR THE DESIGN OF PARALLEL AND DISTRIBUTED ARCHITECTURES FOR RELATIONAL DATABASE QUERY PROCESSORS												
Research Chairman: Thomas C. Hartrum, Ph.D. Associate Professor of Electrical Engineering												
<i>[Signature]</i> Approved for public release: IAW AFPR 190-1 CPT E. W. HALLER Dean for Research and Technical Development Air Force Institute of Technology (AFIT) Wright-Patterson AFB OH 45433												
20. DISTRIBUTION/AVAILABILITY OF ABSTRACT <input checked="" type="checkbox"/> UNCLASSIFIED/UNLIMITED <input type="checkbox"/> SAME AS RPT <input type="checkbox"/> DTIC USERS		21. ABSTRACT SECURITY CLASSIFICATION  <b>UNCLASSIFIED</b>										
22a. NAME OF RESPONSIBLE INDIVIDUAL Thomas C. Hartrum		22b. TELEPHONE (Include Area Code) (513) 255-3576	22c. OFFICE SYMBOL AFIT/ENG									

The design of faster relational database query processors to improve the data retrieval capability of a database was the goal of this research. The emphasis was on evaluating the potential of parallel implementations to allow use of multiprocessing. First, the theoretical considerations of applying relational operations to distributed data was considered to provide an underlying data distribution and parallel processing environment model. Next, analytical models were constructed to evaluate various implementations of the select, project, and join relational operations and the update operations of addition, deletion, and modification for a range of data structures and architectural configurations. To predict the performance of the query processor for all cases, the individual operator models needed to be extended to models for complex queries consisting of several relational operations. The solution to modeling multi-step queries was the use of a general form to express a query. This normal form query tree uses combined operations to express relational algebra equivalent queries in a standard form. This standard tree form was then used to construct analytical models for multi-step queries. These models provide the capability to model the potential of different forms of parallelism in solving complex queries. The results of the analytical models present a logical design for a multiprocessor query processor. This logical query processor using multiple processors and employing parallelism illustrates the potential for an improved query processor using parallel processing when the analytical model results of complex queries are compared to a benckmark of some current database systems.

END

DATE

FILMED

APRIL

1988

DTIC